VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy pro rychlé indexování textu

Algorithms for Fast Text Indexing

2018 Martin Prouza

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Prouza**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Algoritmy pro rychlé indexování textu
Algorithms for Fast Text Indexing

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce bude implementovat metodu Ideal Hash Tree pro indexování textových dat. Dále vytvořit přehled metod, které lze použít pro rychlé indexování textových dat. Výsledkem práce bude porovnat tyto metody s metodou Ideal Hash Tree.

Práce bude obsahovat:
1. Přehled algoritmů pro indexování textových dat.
2. Popis vybraných algoritmů.
3. Návrh implementace a implementaci popsaných metod.
4. Porovnání implementovaných metod.

Seznam doporučené odborné literatury:

[1] Bagwell, Phil. Ideal hash trees. No. LAMP-REPORT-2001-001. 2001.
[2] Crochemore, Maxime, and Wojciech Rytter. Jewels of stringology: text algorithms. World Scientific, 2003.
[3] Gusfield, Dan. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press, 1997.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **doc. Ing. Jan Platoš, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlášení Studenta

„Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal."

V Ostravě Dne 26.4.2018 Podpis

Poděkování

Rád bych poděkoval především panu doc. Ing. Janu Platošovi, Ph.D., který mi poskytl odbornou pomoc, cenné rady a materiály, které jsem využil k napsání této práce. Také mým rodičům a své drahé polovičce za morální pomoc, která mi dodala sílu a motivaci tuto práci dokončit.

Abstrakt

Při práci s textovými daty, je velice častým problémem naleznout rychlou a efektivní metodu pro vyhledávání řetězců. Pod tímto pojmem si můžeme představit různé ověřování duplicit či hledání hodnot pomocí vyhledávacích klíčů. Mnohé z těchto problémů řeší algoritmy založené na hashovacích funkcích.

Hashovací funkce se používají v různých aplikačních oblastech. Mohou to být autentizace informací, porovnávání změn souborů v cloudových řešeních, ověřování integrity zpráv a mnoha dalších.

Cílem této diplomové práce je nalézt a porovnat vhodné algoritmy pro rychlé indexování textu, konkrétněji na hexadecimálních řetězcích (MD5, SHA-1, SHA-512). Při porovnávání se zaměřím na důležité aspekty a omezení daných algoritmů, jejich náročnost při vytváření struktury, vyhledávání řetězců a náročnost na spotřebovanou paměť.

Klíčová slova

Hash Array Mapped Trie, Hashovací funkce, Hashovací tabulka, Dictionary, C#, MD5, SHA-1, SHA-512

Abstract

When we are working with text data, the very common problem is to find fast effective method for finding strings. Under this term, we can imagine various methods for checking duplicates or finding specific values based on search keys. Many of those problems solving algorithms based on hash functions.

Hash functions can be used in various application areas. Those can be information authentication, cloud comparisons in could solutions, checking message integrity and many others.

The focus of this diploma thesis is to find and compare suitable algorithms for fast text indexing, more specific on hexadecimal strings (MD5, SHA-1, SHA-512). When comparing, I will focus on important aspects and constrains of the algorithms, their structural complexity, searching for strings and consumed memory.

Keywords

Hash Array Mapped Trie, Hash funcion, Hash table, Dictionary, C#, MD5, SHA-1, SHA-512

Obsah

1 Úvod

Pro moderní kryptografii jsou hashovací funkce běžným standardem. Smysl a úkol těchto funkcí je ze vstupních dat vytvořit unikátní bitové posloupnosti právě za pomocí hashů. S hashovacími funkcemi se setkáváme v celém našem širokém okolí, aniž bychom si to uvědomovali. Jsou běžně používány v mnoha aplikačních oblastech. Například u kódování hesel, což samo o sobě představuje široké spektrum oblastí využití od přihlašování se k počítači, k elektronické poště, do internet bankingu nebo jako autorizační metoda pro práci s elektronickými podpisy či i u synchronizačních operacích při práci se soubory. Využívání hashů nám přináší výraznou výhodu ve zjednodušení složitějších objektů nebo možnost porovnávat hodnoty, aniž bychom znali jejich skutečné hodnoty.

Cílem práce bude nalézt ideální strukturu pro rychlou práci s textovými řetězci. Jedna z takovýchto struktur je Hash Array Mapped Trie, představená v článku Ideal Hash Trees [1], kterou napsal Phil Bagwell. Úkolem je tedy implementovat tuto strukturu a porovnat ji s již existujícími algoritmy na C# platformě. V našem případě jsem zvolil porovnání HAMT s třídami HashTable a Dictionary, které by měly efektivně pracovat s textovými řetězci a algoritmem na principu stromu s názvem Efficient Text Pattern Search Tree [2].

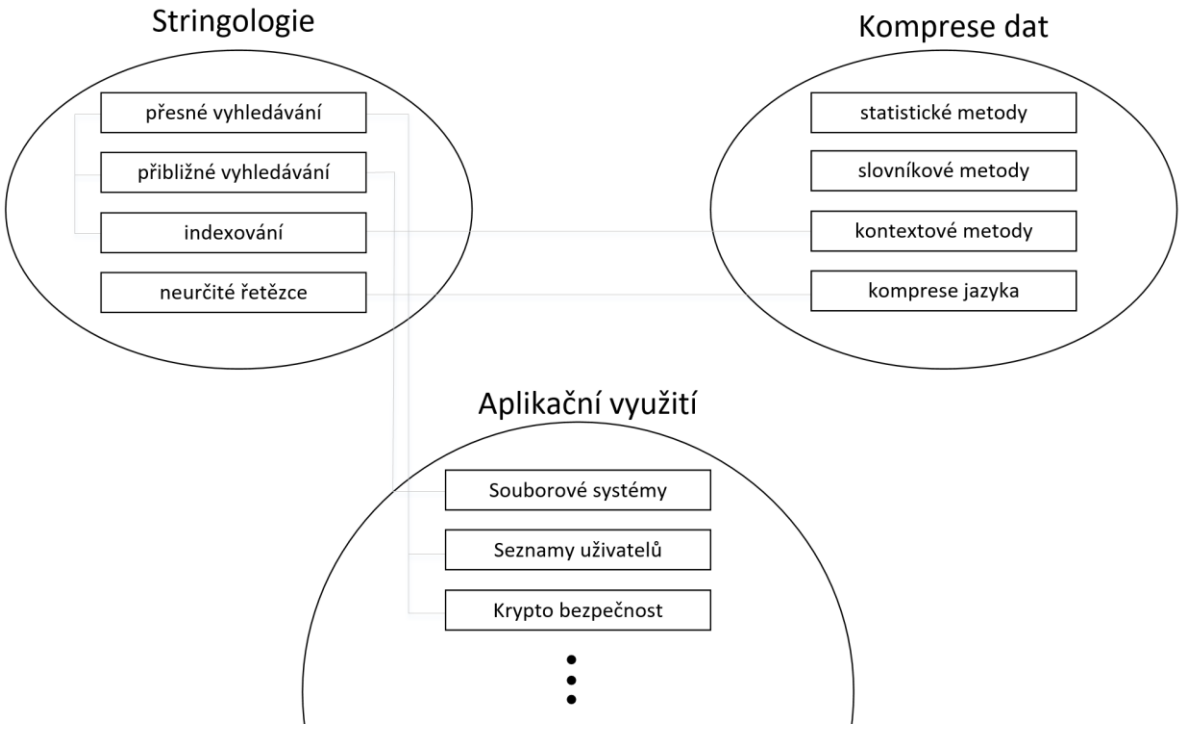
V úvodních kapitolách nastíním problematiku stringologie, vysvětlím, co se pod tímto pojmem skrývá a kde se ním můžeme setkat. Popíšu několik nejzákladnějších principů vyhledávání řetězců v textech a srovnám jejich náročnosti. Z této nauky vycházejí hashovací funkce, se kterými vás posléze seznámím, popíšu k čemu slouží, jak vznikají a vyberu pár nejzákladnějších, které budu využívat pro experimentální část práce.

Rozeberu na jakém principu jsou hashe používány v datových strukturách a v praktické části důkladně rozepíšu princip struktury HAMT, které v experimentální části podrobím testům a srovnám je s jinými strukturami, které jsou v současné době používány na indexování textu.

2 Stringologie

Stringologie je věda o zpracování řetězců a posloupností symbolů [3]. Začala vznikat v 70. letech 20. století, ale samotný pojem této vědy zavedl Zvi Galil až v roce 1984 na konferenci NATO Advanced Research Workshop on Combinatorial Algorithms on Word [4]. V prvopočátcích se první řešené úkoly potýkaly s problémy nalézání vzorků v textech. Postupně vznikaly novější a novější publikace, které opravovaly problémy či zefektivňovaly postupy starších metod. Mnohé z nich fungovaly na principu přibližného vyhledávání, kde se ve výsledku hledání objevovaly vzorky s chybami. Stringologie se pak vyvíjela cestou k úplným indexům, definováním chyb a vyhledávání určitých pravidelností v textech.

Se stringologií je úzce svázána i komprese dat (viz Obrázek 1). Tato věda se zabývá tím, jak co nejefektivněji spořit místem při práci s daty. První statistické metody vyvinuté v 70. letech 20. století se zabývaly pouze statistickým rozdělením jednotlivých symbolů. Symbolům, které byly nejčastěji v daném vzorku nalezeny, byly přidělovány nejkratší substituční kódy. Složitější metody, známé jako slovníkové, pak zohledňovaly časté opakování částí textů. Ty šetřily místo pomocí ukazatelů na předchozí výskyty takových řetězců. Mezi nejpokročilejší se řadí kontextové metody, které využívají kontextu v textu k předvídání následujících symbolů. Také umožnily využívání fulltextového vyhledávání i v komprimovaných datech.



Obrázek 1 - Vztah mezi stringologií, kopresí dat a jejich využití

2.1 Základní pojmy

V této kapitole popíši termíny a symboly, které se používají především k detailnímu popisu jednotlivých funkcí či algoritmů. Vyberu zde pouze ty nejdůležitější, které jsou důležité k pochopení mého zadání. Zbytek pojmů, které nezahrnu, je možné dohledat v použité literatuře.

2.1.1 Symboly

V informatice se pod pojmem symbol rozumí značka, je unikátní, reprodukovatelná, atomická, graficky snadno znázornitelná a nejlépe obecně známá. Spadají pod ně symboly například běžné abecedy (a, b, c, ..., z), ať malém či velké provedení, arabské číslice (1, 2, 3, ...), u kterých můžeme narazit na různé reprezentace soustav, i v kombinaci s abecedou (binární [0,1], desítková [0-9], hexadecimální [0-F]), matematickými symboly ($\pm$, $\times$, $\div$), arabskými, japonskými nebo čínskými znaky nebo symboly řecké abecedy a dalšími znaky.

2.1.2 Abeceda

Konečná neprázdná množina se nazývá abecedou. Značí se velkým písmenem Σ. Čím je abeceda rozsáhlejší tím více nese s sebou informací každý jednotlivý symbol. Abeceda je uspořádaná a každé libovolné dva symboly z ní lze mezi sebou porovnat.

2.1.3 Řetězce

Konečná sekvence symbolů nad danou abecedou se nazývá řetězec. Pokud řetězec neobsahuje žádné symboly, tak ho nazýváme prázdným a značí se ε. Množina všech řetězců na abecedou Σ se značí Σ^* a obsahuje nekonečné množství libovolných řetězců, stejně tak může obsahuje řetězec prázdný. Pokud prázdný řetězec není dovolen, značí se tato abeceda Σ^+.

2.2 Algoritmy pro vyhledávání řetězců

2.2.1 Triviální algoritmus

Zde pominu různé heuristické a deterministické metody a zaměřím se čistě na algoritmický postup vyhledávání. Jestliže máme odpovědět na otázku, zda se řetězec *p* nachází v textu *t*, tak musíme postupovat systematicky. Jedním z nejjednodušších algoritmů, který tuto otázku zodpoví, je algoritmus naivní neboli triviální. Ten spočívá v přiložení vzorku řetězce *p* k začátku textu *t* a porovná ho na odpovídajícím úseku textu. Pokud řetězec nebyl nalezen, tak se algoritmus systematicky posouvá o jeden symbol podle směru průchodu a proces opakuje, dokud hledaný řetězec *p* nebyl nalezen nebo se již není kam posouvat v rámci textu. Algoritmus tohoto algoritmu si můžete prohlédnout níže (Kód 1 - Triviální algoritmus hledání řetězce)

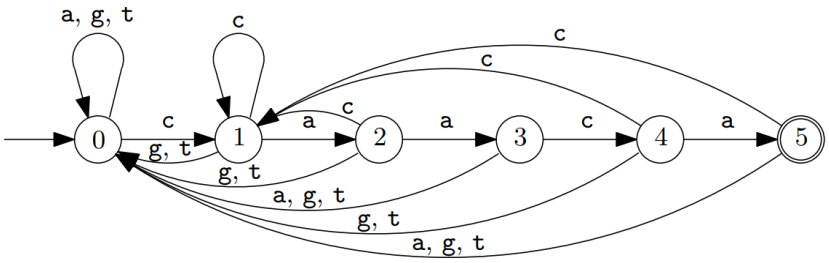
```
Search(textFile t, string p)
{
  if(t.length > p.length)
  {
    i = 0
    while( i <= t.length - p.length)
    {
      j = 0
      while(j <= p.length && p[j+1] == t[i+j+1])
      {
        j++
        if(j == p.length)
        {
          print "Řetězec byl nalezen na pozici " i+1
        } else
        {
          i++
        }
      }
    }
  }
}
```

Kód 1 - Triviální algoritmus hledání řetězce

Složitost tohoto triviálního algoritmu bude určena dvěma cykly v těle algoritmu. Jeden cyklus je závislý na délce textu a druhý na délce hledaného řetězce. V nejhorším případě tak bude nutné provést celkem |řetězec|×|text| operací porovnání. Pro operaci porovnání je potřeba konstantního času, stejně tak i pro ostatní části řízení programu. Celková složitost tohoto algoritmu tedy bude $O(m \times n)$, kde n značí délku textu a m délku hledaného řetězce.

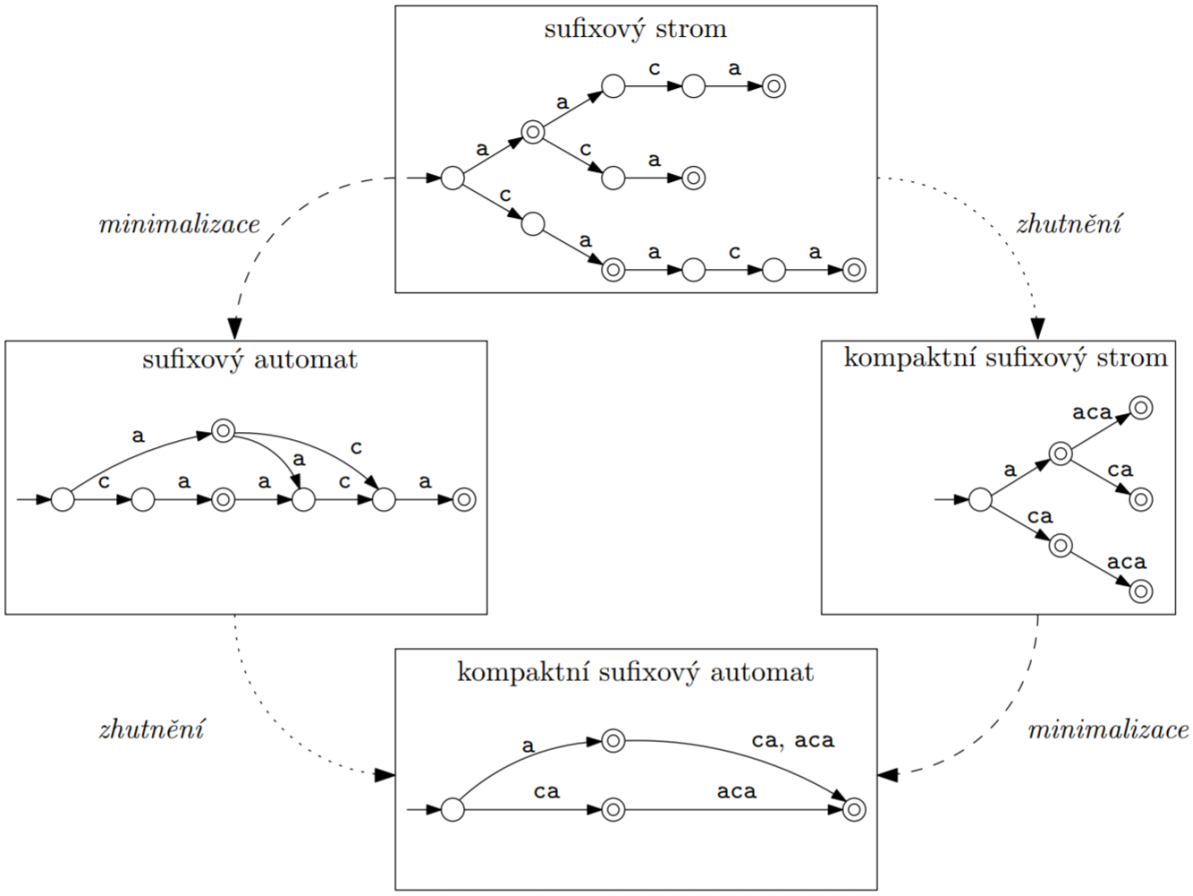
2.2.2 Přesné vyhledávání

Prvními úlohami stringologie bylo nalézání přesného výskytu vzorku $p = p[1..m]$, délky m, v textu $t = t[1..n]$ délky n, kde p i t jsou řetězce nad abecedou Σ (značíme jako $p,t \in \Sigma^*$). Tento problém je možné vyřešit užitím mnoha různých algoritmů. Demonstruji tento postup na deterministickém konečném automatu (DKA) (viz Obrázek 2)



Obrázek 2 - Deterministický konečný automat pro hledání řetězce p = caaca, Σ={a, c, g, t} [5]

Tento DKA je definován jako uspořádaná pětice $M = (Q, \Sigma, \delta, q_0, F)$, kde Q je konečná množina stavů, Σ je konečná vstupní abeceda, δ je přechodová funkce $Q \times \Sigma \mapsto Q, q_0 \in Q$ je počáteční stav a $F \subseteq Q$ je množina koncových stavů. Tento automat přijímá na vstupu řetězec, ten zpracuje přechodem do koncového stavu a hlásí jednotlivé výskyty řetězce *p* v textu *t*.



Obrázek 3 - Příklady algoritmů pro vyhledávání řetězce [5]

2.2.3 Úplný index

Další možnost, jak vyhledávat přesnou posloupnost řetězců v textu je předzpracování samotného textu. Nad textem *t* se vytvoří úplný index a řetězec *p* lze poté vyhledat v čase *O(n)*, což nám může napovědět,

že tato náročnost není přímo závislá na délce textu *t*. Pro demonstraci je možné využít například sufixový strom (Suffix Trie, ST). Jedná se o deterministický konečný automat, který přijímá všechny přípony neboli sufixy daného textu. Také dokáže rozpoznat i možné podřetězce daného textu. Pokud existuje cesta z počátečního stavu u řetězce *r* do libovolného stavu v sufixovém automatu, pak je řetězec *r* podřetězcem textu *t*.

Pokud tento postup sufixového stromu minimalizujeme, získáme sufixový automat (viz Obrázek 3). Tento ST je minimální deterministický automat přijímající všechny přípony textu *t*. Na druhou stranu, zhutněním ST získáme kompaktní sufixový strom, ve kterém jsou přechody ohodnoceny přímo pomocí řetězců místo jednotlivých symbolů. Každá posloupnost přechodů, která se nevětví a neobsahuje koncové stavy, se při procesu zhutnění nahradí jedním přechodem ohodnoceným řetězcem složeným ze symbolů ohodnocující původní posloupnost přechodů. V obou případech sufixových automatů můžeme získat kompaktní sufixový automat.

Pokud v sufixovém automatu označíme všechny stavy jako koncové a eliminujeme všechny stavy, které nepatří do kostry automatu, získáme „faktorový oracle“ [1]. Tento automat má méně stavů než minimální deterministický automat přijímající všechny podřetězce textu *t*, tak ztrácíme informaci o vrácení podřetězce *p* z textu *t*. Nastávají zde problémy tykající se ověřování hypotéz, kdy získáme jednoznačnou odpověď, že *p* není podřetězec textu *t*, ale naproti tomu nezískáme odpověď *p* je podřetězec textu *t*. Pouze informaci že *p* může být podřetězcem textu *t*, a proto je tuto hypotézu nutné ověřit.

Tímto nám vznikne sufixové pole pro text *t*, což je pole indexů identifikujících všechny přípony textu *t* seřazené lexikograficky. Sufixové pole má sice náročnost konstrukce v čase *O(n)*, ale v praxi bývá nejrychlejším algoritmem běžícím v čase $O(n^2 \log n)$ [6].

	čas vyhledání	čas konstrukce	max. velikost
sufixový strom	$O(m)$	$O(n^2)$	n^2
kompaktní sufixový strom	$O(m)$	$O(n \log \lvert\Sigma\rvert)$	$2n$
sufixový automat	$O(m)$	$O(n \log \lvert\Sigma\rvert)$	$2n - 1$
kompaktní sufixový automat	$O(m)$	$O(n \log \lvert\Sigma\rvert)$	$n + 1$
faktorový oracle	$O(m)$	$O(n \log \lvert\Sigma\rvert)$	$n + 1$
sufixové pole	$O(m+\log n)$	$O(n^2 \log n)$	n

Tabulka 1 - Časové a paměťové náročnosti algoritmů

Tabulka 1 ukazuje přehled základních indexovacích struktur s časy vyhledávání, konstrukce a maximální velikostí, která udává maximální počet stavů automatu neboli maximální velikost sufixového pole.

[1] anglicky factor oracle [14]

3 Hashovací funkce

3.1 Definice

Principy hashovací funkce se chápe jako zobrazení h, které přiřazuje různě dlouhé vstupní posloupnosti či objektu, podle jasně definovaného algoritmu, výstupní posloupnost výhradně pevné délky. Tato posloupnost, která je charakterizující pro daný objekt je označována jako hash, což můžeme přeložit jako otisk objektu.

Základní vlastnosti pro hashovací funkce:

- Pro libovolné množství dat je výstupní hash vždy stejné délky.
- I minimální změna na vstupních datech má za následek velice rozdílný výsledek na výstupu funkce (na první pohled se hashe od sebe budou lišit)
- Pokud dva objekty mají stejný hash, je zde vysoká pravděpodobnost, že jsou tyto objekty stejné. (pravděpodobnost kolizí s rostoucím počtem objektů stoupá)
- Z výstupní hodnoty hashe je prakticky nemožné získat zpětně vstupní hodnotu

Formálně se hashovací funkce dá popsat jako matematická funkce h, která převádí vstupní posloupnost d bitů (bytů) libovolné délky na posloupnost pevné délky r bitů (bytů).

$$h: D \rightarrow R, kde\ |D| > |R|$$

Z této definice se dá vyčíst, že umožňuje existenci kolizních stavů. Znamená to, že pro dvojici vstupních dat $(x, y), x \neq y$ takových, že $h(x) = h(y)$. Vyjádřeno slovně, pro dvojici o různých vstupních hodnotách může být spočtena stejná hash.

I když kolizní stavy jsou nežádoucím prvkem hashovacích funkcí, tak se jim nelze úplně vyhnout. Můžeme pouze na základě propracovanějších metod výrazně omezit jejich existenci. Cílem je tedy dosáhnout co nejvyšší pravděpodobnosti, že dva objekty se stejnou hash hodnotou jsou totožné. Kolizní stavy v praxi nemohou být plně eliminovány, neboť objekty o dané délce by musely být identifikovány stejně dlouhým hash kódem, a tím bychom přišli o hlavní výhodu hashovacích funkcí, kterou je komprese dat pro jejich následné zpracování.

3.2 Vlastnosti funkcí

V obecném pohledu se hashovací funkce vyznačovaly tím, že libovolnému vstupu přiřazovaly výstupy o pevné délce. V současnosti se tento pojem používá především v kryptografii a to zejména pro svou bezpečnostní funkci. Unikátnost procesu hashování totiž přináší důležité vlastnosti jednosměrnosti procesu a rezistence.

Ideální kryptografická hashovací funkce je taková, která zajistí jednoznačnou autentizaci, zaručení integrity a zpětnou neprolomitelnost procesu. Taková funkce samozřejmě neexistuje, proto je cílem vymyslet takovou hashovací funkci, která se těmto ideálům přiblíží co nejvíce.

Vlastnosti různých hashovacích funkcí určují obtížnost jejího napadení. Tím se rozumí výpočetní složitost dle aktuálních technologických možností v reálném čase. Využiji zde citace Jaroslava Pinkavy [7], který popisuje různé druhy vlastností hashovacích funkcí:

V1. Praktická efektivnost: Pro dané *x* je výpočet *h(x)* efektivně proveditelný (přesněji, je proveditelný v čase, který je omezen polynomiální funkcí délky vstupu *x*).
V2. Mixující zobrazení: Pro každý vstup *x* má výstupní hodnota „náhodný" charakter
V3. Rezistence vůči kolizím: Je z výpočetního hlediska neuskutečnitelné nalézt dva vstupy $(x, y), x \neq y$,aby $h(x) = h(y)$.
V4. Rezistence prvního vzoru: Pro danou hodnotu hashe *h* je výpočetně neuskutečnitelné nalézt vstupní řetězec *x* tak, že $h = h(x)$.
V5. Rezistence druhého vzoru: Je výpočetně neuskutečnitelné pro daný vstup *x* nalézt druhý vstupní řetězec *z* tak, že $h(x) = h(y)$. Tato vlastnost se od vlastnosti V3 liší tím, že zde je jeden vstup již fixován.
V6. Rezistence vůči blízkým kolizím: Je z výpočetního hlediska neuskutečnitelné nalézt dva vstupy $(x, y), x \cong y$ tak, že *h(x)* a *h(y)* se liší jen v malém počtu bitů. Blízké kolize jsou nejjednodušším příkladem zakázaných vztahů mezi výstupy hashovací funkce.

Na základně prvních pěti pravidel, můžeme definovat dva typy hashovacích funkcí:

1. OWHF (One-Way Hash Function)
2. CRHF (Collision Resistant Hash Function)

3.2.1 OWHF – One-Way Hash Function

Jedná se o jednosměrnou hashovací funkci. Splňuje výše uvedené vlastnosti V1-V5. Takovou funkci je možné jednoduše vyčíslit, ale je téměř nemožné se výsledku funkce dobrat zpátky na začátek procesu. Zpětně tedy nelze zjistit co bylo na vstupu funkce. Definice tedy zní, že ze zadaného *x* lze snadno získat *f(x)*, ale výpočet funkce inverzí, tedy získání hodnoty *x* při znalosti *f(x)*, je prakticky neřešitelný.

3.2.2 CRHF – Collision Resistant Hash Function

Tato hashovací funkce je rezistentní vůči kolizím. Z uvedených vlastností splňuje podmínky V1-V3. Hashovací funkce rezistentní vůči kolizím je vždy jednosměrnou hashovací funkcí. První tvrzení můžeme dokázat jednoduše, neboť z vlastnosti V3 plyne vlastnost V5. Jednosměrnost funkce rezistentní vůči kolizím se dá dokázat pomocí techniky dokazování sporem. To se provede tak, že předpokládaný výrok znegujeme a pokusíme se dojít k nějakému sporu, který by nám tuto negaci popřel.

3.3 Kolizní stavy

Jeden z hlavních aspektů, na který by se měl u dobré hashovací funkce brát zřetel, je problém řešení kolizí nebo lépe předcházet jim. Tato problematika u kryptografických funkcí řeší pomocí komplexního

postupu od zesílení řetězce pomocí přídavných bitů, přes řešení pomocí blokových šifer. Kolize, jak jsem již výše zmínil, vnikají, když $h(x) = h(y)$ a zároveň $x \neq y$. V takovém případě dva různé objekty ukazují na stejné místo v hashovací tabulce. Jak často ale k samotným kolizím dochází? Jak velká musí být množina objektů, aby v ní s nezanedbatelnou pravděpodobností existovaly dva různé objekty se stejnou vlastností? Tuto otázku řeší jeden z nejznámějších problémů kolizí, který se nazývá narozeninový paradox.

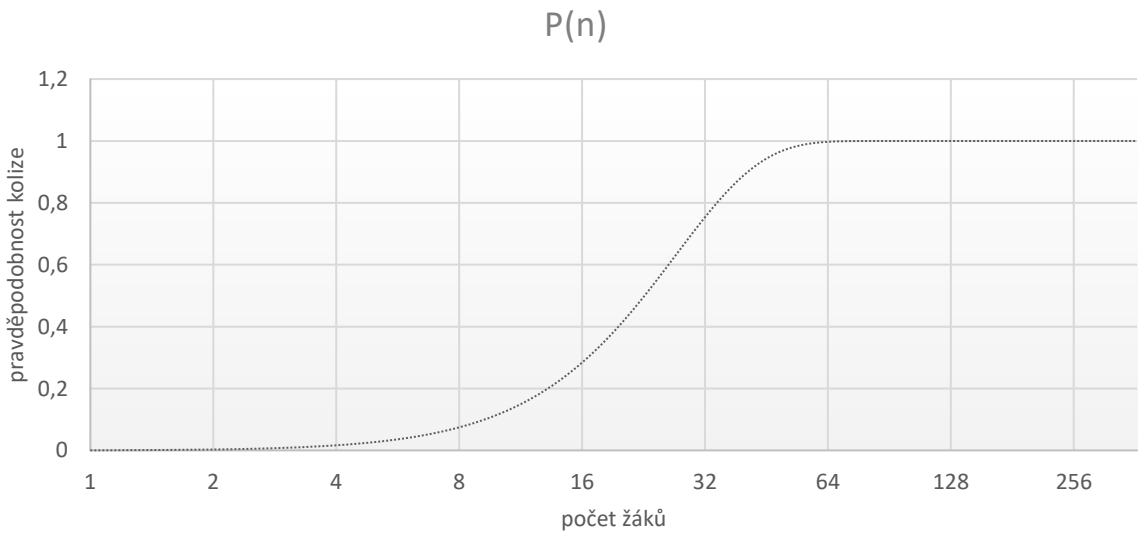
3.3.1 Narozeninový paradox

Představme si množinu *M* o počtu *n* objektů. Kolik z těchto objektů musíme vybrat, abychom v množině objektů obrazů získali kolizi? Tento problém je běžně známý pod názvem „Narozeninový paradox“, jmenuje se podle jednoduše představitelného přirovnání narozenin žáků v jedné třídě.

Mějme ve třídě množinu žáků a množinu obrazů, která je představena právě narozeninami. Otázka zní, kolik žáků musí ve třídě minimálně být, aby se s v této množině *n* prvků nalezli dva takoví, kteří slaví narozeniny ve stejný den. Rovnicí můžeme tento problém snadno vyčíslit.

$$p(n) = 1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{n-1}{365}\right) = \left(\frac{365 \cdot 364 \cdots (365 - n + 1)}{365^n}\right) = \frac{365!}{365^n\,(365 - n)!}$$

Druhý žák nemůže mít narozeniny ve stejný den jako první (364/365), třetí je nemůže mít ve stejný den jako první dva žáci (363/365), atd. Hodnota překročení pravděpodobnosti 0,5 nastává při $n = 23$ (okolo 50,7%). Níže uvedený Graf 1 zobrazuje průběh pravděpodobnosti kolizí narozenin vzhledem k počtu žáků.



Graf 1 - Znázornění pravděpodobnosti výskytu kolize vzhledem k počtu žáků

3.3.2 Řešení kolizí

3.3.2.1 Zřetězené hashování

U zřetězeného hashování, když dojde ke kolizi dvou nebo více klíčů, které ukazují na stejné místo v tabulce se stejnou adresou, tak se vytvoří na této adrese zřetězený seznam kolidujících klíčů. Tyto klíče se nacházejí mimo hlavní tabulku v dynamické paměti. Aby byl zachován konstantní čas vkládání záznamu do tabulky, tak se poslední přidaný klíč umístí na začátek kolizního seznamu.

3.3.2.2 Otevřené adresování s lineárním vkládáním

Při kolizním stavu spočteného hashe ze vstupu x funkcí $h(x)$ s adresou, kam má být daný hash uložen, znamená, že místo v tabulce je již obsazeno. Použije se tedy metoda založená na sekvenčním průchodu, než se nalezne volné místo v hash tabulce. Určí se místo v paměti na místě $h(x)+p$, kde p je pozice, která je postupně zvyšována v závislosti na obsazené paměti. Musí být ale zajištěno, že hash tabulka nebyla již zcela zaplněna. V takovém případě by nám při pokusu uložení další kolizní hodnoty a nalezení volného místa v tabulce vznikl nekonečný cyklus průchodu.

Nevýhoda této metody spočívá v nerovnoměrném ukládání hodnot do hash tabulky. V případě kolize se tvoří dlouhé shluky obsazených pozic v tabulce, a to přispívá k rychlejšímu výskytu dalších kolizí. Tato metoda tedy není vhodná pro ukládání hodnot, neboť při ní nedochází k rozprostření hash hodnot napříč úložištěm.

3.3.2.3 Otevřené adresování s dvojitým hashováním

Tato metoda je vylepšením algoritmu předchozího. Pomocí dvojitého hashování odstraňuje nedostatky lineárního vkládání. Pokud tedy dojde při procesu vkládání klíče ke kolizi, tak se nepřidá tento klíč do tabulky na volnou pozici, ale je spočtena nová úroveň hashovací funkce. V takovém případě si ale musíme poznamenat, že na právě tomto místě tabulky dochází ke koliznímu stavu víro klíčů a je nutné pro hledanou hodnotu přepočítat hodnotu hashe sekundární funkcí, jinak bychom uložený klíč již zpět nezískali. Princip sekundární hashovací funkce by měl být zvolen rozumně tak, aby druhá funkce kolizní klíče odlišila.

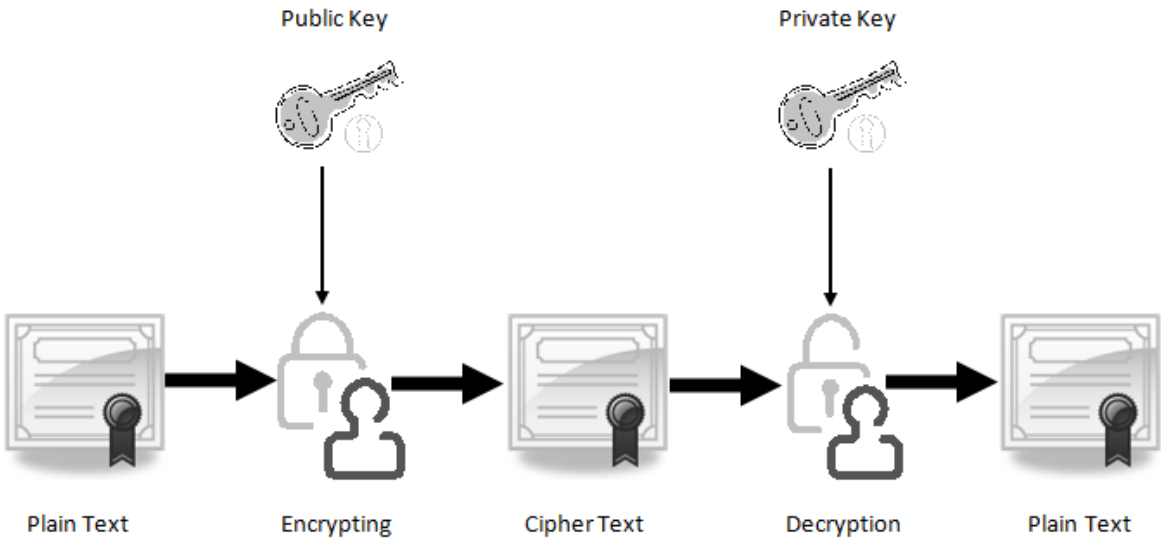
3.4 Využití hashovacích funkcí

Díky svým vlastnostem se hashovací funkce využívají v různých oblastech informačních technologií. Důvody jejich použití jsou především snadná identifikace objektů s nízkou mírnou pravděpodobnosti kolizí. Slouží tak především jako jedinečný identifikátor o přesně stanovené délce. Uvedu zde několik příkladů, kde si hashovací funkce nalezly svá uplatnění.

3.4.1 Ověřování integrity

Ověřování integrity neboli celistvosti zprávy znamená proces rozhodnutí, zda přijaté informace jsou jednoznačně shodné s informacemi odeslanými. Takto se dozvíme, zda se zprávou bylo jakýmkoli způsobem cestou modifikováno. S tímto úzce souvisí i kódy na detekčních a korekčních principech. Smysl detekčních kódů je v odhalení chyb při přenášení informací. Nejčastěji toho docílíme vypočítáním kontrolního součtu[2] odesílané zprávy, který k této zprávě přiložen. U příjemce se následně použije stejná funkce kontrolního součtu a pokud jsou stejné, tak integrita zprávy narušena nebyla.

V praxi se s podobným postupem můžeme setkat u autorizace přenášených zpráv pomocí veřejných a osobních klíčů[3].



Obrázek 4 - Princip použití kryptovacích klíčů [8]

3.4.2 Autentizace informací

Tento proces se zabývá ověřováním proklamované identity objektu, známý pod pojmem digitálních či elektronických podpisů. Proces ověřování rozsáhlé zprávy může být někdy velice složitý, proto pro zjednodušení se používají právě autentizační principy. Místo kódování a kontrolování celé zprávy se ověřuje pouze autentičnost její hashe. Zpráva je tedy zasílána nezabezpečená. Zabezpečuje se pouze její hash.

3.4.3 Porovnávání souborů

Své místo si hashovací funkce nalézají i u porovnávání souborů. Nepracují na základě znalosti o obsahu souborů, ale otisku jejich vlastností. Tím se snadno jednotlivé soubory od sebe dokáží odlišit, bez toho,

[2] Jinak známý jako Message Integrity Code
[3] Z anglického překladu public and private keys

aniž bychom museli zpracovávat obsah celého souboru, což by obzvláště u velice objemných souborů mohlo hrát výraznou roli v efektivitě použití. Tento proces verifikování souborů se může uplatnit v celé škále aplikačních programů. Při procesech zálohování souborů, které se nevyskytují na bezpečném médiu nebo byly pozměněny, přes systém synchronizace cloudových úložišť až po vyhledávání souborů v P2P sítích[4].

3.4.4 Správce hesel

Jeden z hlavních bezpečnostních rizik je práce s hesly. Spočívá v tom, že kdyby se nevyžádaná osoba dostala k procesu autentizace hesla a tato hesla se porovnávala tak, jak jsou (z angl. *as is*), Vzniká bezpečností riziko úniku hesla ze služby a získání jeho skutečné hodnoty. Takto získané heslo se může útočník pokusit využít i v jiných službách a pokud poškozený uživatel používá stejné heslo pro všechny služby, má o starosti postaráno.

Při použití hashovacích funkcí na heslo, získáme unikátní identifikátor hesla, který následně uložíme k danému uživateli a kdykoli nově přijde požadavek o autentizaci daného uživatele, už se porovnává pouze uložený identifikátor s příchozím heslem, ze kterého se opět spočítá hash podle stejné metody. Tímto nám odpadá problém, že ve službě máme fyzicky uloženou skutečnou podobu hesla. Z vlastností hashovacích funkcí totiž víme, že proces hashování je jednosměrný, a tak je prakticky nemožné získat zpětně z hashe původní podobu zadaného hesla.

Snadno si tak můžete vyzkoušet, jak bezpečně internetové služby zachází s vašimi přihlašovacími údaji. Pokud totiž zapomenete své heslo, tak služba, která zná pouze spočtenou hash hodnotu vašeho hesla, vám jej nikdy nemůže už zpětně zaslat zpátky, třeba do emailové schránky. Budete si muset vytvořit heslo nové.

3.4.5 Hashovací struktury

Nejběžnější a nejznámější hashovací strukturou je hashovací tabulka. Funguje na principu adresování hashovaných klíčů do tabulkové struktury. Používá se především pro rychlé dohledávání položek v poli nebo jiné homogenní[5] kolekci.

Pomocí hashovací funkce se přiřazuje hodnotě klíče index neboli ukazatel do datové struktury. Samotný klíč je tedy přímo používán pro zjištění adresy tohoto klíče v rámci této struktury. Existuje mnoho různých strukturálních alternativ, které fungují na podobném principu jako hashovací tabulka nebo z ní přímo vycházejí. Jedním z nich je například Hash Array Mapped Trie, který je tématem této práce a podrobněji se mu budu věnovat v následující kapitole. Algoritmy založené na hashovacích funkcích mají zpravidla nejhůře složitost konstantní.

[4] Peer to peer sítě. Systém vzájemného sdílení mezi několika uživateli najednou bez nadřízeného prvku (obecně server).

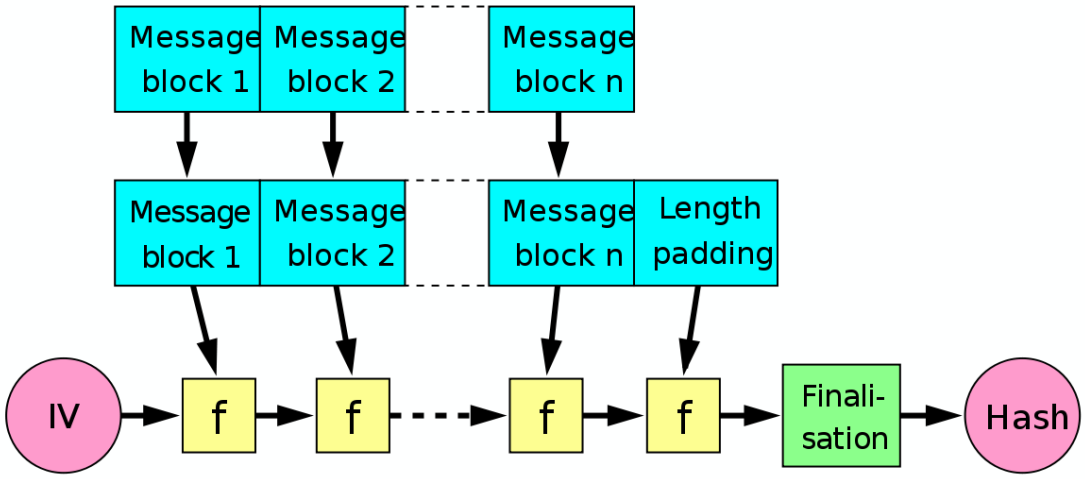
[5] Homogenní strukturou se rozumí taková, která má všechny položky stejného typu.

3.5 Základní kryptografické hash funkce

Kryptografické hash funkce jsou různých velikostí a tvarů. V zásadě existují dvě hlavní kategorie hash funkcí. Hash funkce, které závisí na klíči pro jejich výpočet, obecně známé jako Message Authentication Code neboli MAC a hash funkce, které na klíči nezávisí. Obecně známé jako „un-keyed hash functions“ zjednodušeně hash funkce. Všechny známé hash funkce jsou založené buď na šifře nebo na modulární aritmetice.

3.5.1 Merkle-Damagard konstrukce, MD

Pojmenované po dvou zakladatelích, Američanovi Ralphu C. Merklovi a Dánovi Ivanu Demagardovi. MD struktura definuje krok za krokem proces pro dosažení fixní délky výstupního řetězce z libovolně dlouhé vstupní hodnoty.



Obrázek 5 - Princip MD hash funkce[6]

Základní prvky konstrukce MD hashe jsou:

- I.V.: Inicializace vektoru nebo počáteční hodnota jako pevná hodnota použitá jako řetězová proměnná pro první iteraci.
- F: Funkce komprese neboli jednosměrná funkce, která je buď speciálně navržena pro hashování, anebo je navržena na základě blokových šifer. Funkce komprese zpravidla pracuje na vstupu s pevnou délkou a produkuje výstup s pevnou délkou.
- Finalizace: Funkce transformace výstupu, která obvykle snižuje délku výstupní hodnoty v posledním kroku iterace.
- Hash: Otisk zprávy (Message Digest) nebo výsledná hash.

[6] Obrázek převzat z https://en.wikipedia.org/wiki/Merkle%E2%80%93Damg%C3%A5rd_construction

Jak si můžeme všimnout (viz Obrázek 5), zpráva, která má být zahashována, je nejprve rozdělena na *n* bloků stejné délky. Skutečná délka bloků záleží na požadavcích nastavených kompresních funkcí *f*. Následně postupně zakládána zpět dohromady pomocí různých operací, finalizována a vznikne výsledná hash.

3.5.1.1 MD2

Jedná se o první publikovaný algoritmus funkcí MD určený pro 8bitové procesory s malou pamětí. MD2 byla vyvinuta speciálně pro embedded zařízení. Byla navržena v roce 1989 Ronaldem Rivestem a publikována v dubnu 1992 jako RFC 1319.

Hashování probíhá celkově ve třech fázích.

F1. V první fázi je řetězec, který má být hashován, zarovnán na délku dělitelnou 16.
F2. V druhé fázi se spočítá 16 bytový kontrolní součet, který je připojený ke konci zarovnané zprávy. Ta je následně rozdělena do 16 bytových bloků.
F3. Fáze třetí se skládá z opakování kompresní funkce, která počítá nové hodnoty pro zřetězení. Zřetězuje se vždy hodnota aktuální proměnné a blok zprávy, který je na řadě. Inicializační hodnota proměnné, která je v každém cyklu aktualizována, je pevná a je součástí algoritmu. Výstupní klíč má délku 128 bitů.

V roce 1997 byl na tuto kompresní funkci popsán útok pány Rogierem a Chauvaudem. Ten nebyl rozšířen na celý algoritmus. Vedlo to k tomu, že v roce 2004 byl demonstrován útok s časovou složitostí 2^{104} a od té doby již tato hashovací funkce není považována za bezpečnou. [9]

3.5.1.2 MD3

Jelikož algoritmus nebyl nikdy publikován, nedočkal se ani praktického využití a byl okamžitě nahrazen algoritmem MD4.

3.5.1.3 MD4

MD4 byl navržen v říjnu roku 1990 profesorem Ronaldem Rivestem z univerzity MIT. Pracuje rychleji než MD2 a používá již 32 bitové operace. Návrh tohoto algoritmu ovlivnil i jiné hashovací funkce, jako jsou např. MD5, SHA a RIPEMD. Algoritmus se využíval k výpočtu hashe hesla v systémech Microsoft Windows NT, XP a Vista. Pro novější verze operačních systému Microsoft byl ale málo bezpečný, a tak se přešlo na novější generaci. Varianta tohoto algoritmu se používala k poskytování jedinečných identifikátorů pro soubory ve výměnných sítích jako byly eDonkey, eMulea. Dříve byl tento algoritmus použit i v protokolu rsync[7].

Na vstupu přijímá funkce blok dat o velikosti 2^{64} bitů. Vstup je nejdříve zarovnán a poté rozdělen na bloky o délce 512 bitů. Algoritmus pracuje s bloky dat o délce 128 bitů, rozdělenými na čtyři stejně dlouhá bitová slova. V každém cyklu hashovací funkce je zpracováván 512 bitový vstup. K výpočtu je

[7] rsync je počítačový program původně vyvinutý pro unixové systémy, který synchronizuje soubory a adresáře mezi různými lokacemi za použití co nejmenšího přenosu dat.

využito tří cyklů, kde každý z nich obsahuje 16 podobných matematických operací, založených na nelineární funkci f, bitovém exkluzivním součtu a levé bitové rotaci. Nakonec je vyprodukovaná hash o délce 128 bitů.

Slabina algoritmu MD4 byla demonstrována Danem Boerem v roce 1991. První kolizní útok byl proveden v toce 1996 a v roce 2004 našel X. Wang velmi efektivní způsob útoku kolizí na celou řadu hashovacích funkcí postavených na konceptu MD4. Později byl útok ještě zdokonalen týmem profesora Sasakiho a generování kolizí se tak stalo snadným postupem jako jejich ověřování. [10]

3.5.1.4 MD5

Jedna z nejpoužívanějších hashovacích funkcí, navržena již v roce 1991 jako náhrada za MD4 a publikována v dubnu roku 1992. Algoritmus se široce využívá ve spoustě různých aplikací.

Funkce má podobnou konstrukci jako funkce MD4. Na vstupu přijímá vstupní data do velikosti 2^{64} bitů. Vstup je zarovnán a rozdělen na bloky dat o velikosti 512 bitů. Stejně jako předešlá funkce, pracuje i tato s bloky dat o délce 128 bitů rozdělenými do čtyř 32 bitových bloků. K výpočtu je zde použito 4 cyklů, kde každý cyklus má 16 operací, které jsou založeny na nelineární funkci f, exkluzivním bitovém součtu a levé bitové rotaci. Výsledná hash má stejně jako MD4 délku 128 bitů.

První útok na hashovací funkci MD5 byl proveden v roce 1993. Byl publikován výsledek hledání pseudo-kolizí na MD5 kompresní funkci, kde dva rozdílné inicializační vektory mohou produkovat stejnou hash. O tři roky později profesor Dobertin oznámil kolizi kompresní funkce MD5. I když nešlo o útok na celou hashovací funkci, bylo doporučeno používat alternativní hashovací funkce jako Whirpool, SHA-1 a RIPEMD-160. V roce 2004 byly nalezeny další nedostatky , které ještě více zpochybnily použitelnost MD5. Jeden z nejefektivnějších útoků popsal český kryptolog V. Klíma, který v roce 2006 vytvořil algoritmus, který i na běžně dostupných strojích byl schopen vyhledat kolize do jedné minuty. Tuto metodu nazval tunelování.

3.5.2 SHA funkce

3.5.2.1 SHA-0 a SHA-1

První specifikace algoritmu SHA-0 byla zveřejněna v roce 1993 jako Secure Hash Standard, FIPS PUB 180. Tuto specifikaci zveřejnila agentura NIST (National Institute of Standards and Technology). Tato specifikace byla krátce po své publikaci označena NSA (National Security Agency) za nedostatečně bezpečnou a byla v roce 1995 nahrazena dalším standardem FIPS PUB 180-1, který je známý pod zkratkou SHA-1. Od SHA-0 se liší jen jednou bitovou rotací v kompresní funkci. Tuto opravu navrhla NSA, aby byla plně zachována kryptografická bezpečnost šifry, ale nepublikovala žádnou zprávu či bližší vysvětlení o slabině hashovací funkce. Zanedlouho byla nahlášena slabina i v upravené hashovací funkci SHA-1. Funkce jsou založeny na podobných principech jako MD4 a MD5.

Oba algoritmy produkují klíč o velikost 160 bitů, který je vytvořen ze zprávy o nejvyšší přípustné délce 2^{64}-1 bitů. Vstup je nejprve zarovnán, rozdělen do bloků o délce 512 bitů. Funkce pracuje s bloky o velikosti 160 bitů, které jsou rozděleny na 5 bitových slov, tedy o velikosti po 32 bitech. Algoritmus

v každém bloku zpracovává 512 bitový vstup, pomocí kterého upravuje kontext. Vstupní blok je v každém cyklu rozšířen na 80x32 bitových slov. Každý blok se skládá ze 4 cyklů, kde každý obsahuje 20 podobných operací založených na nelineární funkci *f*, exkluzivním bitovém součtu a levé bitové rotaci. [11]

3.5.2.2 SHA-2

Institut NIST publikoval další čtyři hashovací funkce v rodině SHA. Všechny jsou označovány zkratkou SHA-2, ale nebyly nikdy standardizovány. Jednotlivé varianty označené podle bitové délky jejich hashí jsou SHA-224, SHA-256, SHA384 a SHA-512. Poslední tři byly navrženy a schváleny v roce 2001 jako standard FIPS PUB 180-2 a veřejně publikovány jako oficiální standard v roce 2002. V únoru roku 2004 byla specifikace doplněna o funkci SHA-224, která měla splňovat kritéria pro použití v šifře TripleDES. Tyto funkce byly patentovány a uvolněny pod bezplatnou licencí.

Hashovací funkce SHA-256 pracuje s 32 bitovými slovy, funkce SHA-512 s 64 bitovými. Při výpočtu využívají různou velikost posunu a počet cyklů, jinak jsou téměř totožné. Další dvě funkce SHA-224 a SHA-384 jsou pouze zkrácenými verzemi prvních dvou funkcí.

Funkce SHA-256 resp. SHA-224 přijímá na vstupu blok dat o velikost 2^{64} bitů. Vstup je zarovnán a rozdělen na bloky o velikosti 512 bitů. Algoritmus pracuje s bloky o velikosti 256 bitů, které jsou rozděleny do osmi bitových slov. V každém bloku je zpracován 512 bitový vstup, jímž je modifikován kontext a k výpočtu je využito 80 podobných operací založených na nelineárních funkcích. Podobně je tomu i u funkcí SHA-512 resp. SHA-384, které se liší tím, že přijímají na vstupu data o velikosti 2^{128} bitů, používají 64 bitová slova, jiné nelineární funkce, konstanty a inicializační vektory.

U funkcí SHA-2 dochází k daleko komplikovanějšímu výpočtu a expanzi vstupního bloku zprávy, než je tomu u funkcí SHA-1 či MD5. Dosud nejsou známé žádné úspěšné útoky na tyto funkce, a tudíž jsou považovány za bezpečné.

	VELIKOST V BITECH	**VELIKOST V BYTECH**	**DÉLKA ŘETĚZCE**
MD2-MD5	**128**	**16**	**8**
MD6	**až 512**	**až 64**	**až 32**
SHA-1	**160**	**20**	**10**
SHA-224	**224**	**28**	**14**
SHA-384	**384**	**48**	**24**
SHA-512	**512**	**64**	**32**

Tabulka 2 - Srovnání velikostí vybraných hash funkcí

3.6 Další kryptografické hashovací funkce

3.6.1 RIPEMD

Jedná se o rodinu hashovacích funkcí navrženou H. Dobbertinem, A. Bosselaersem a B. Preneelem. Zkratka RIPEMD znamená Race Integrity Primitives Evaluation Message Digest. Původně vyvinutá v roce 1992 a její algoritmus byl velice podobný hashovací funkci MD4. Tato původní funkce produkovala výstup o délce 128 bitů.

RIPEMD-128 byla navržena kvůli komptabilitě s aplikacemi, které pracují s délkou klíče 128 bitů. Už v té době byly známy slabiny hashovacích funkcí s takovou délkou klíče a byl doporučen přechod na 160 bitovou verzi algoritmu. RIPEMD-160 měla nahrazovat již nedostatečné 128 bitové hashovací funkce, jako byly MD4 a MD5. RIPEMD-256 a RIPEMD-320 jsou rozšířením těchto funkcí a mohou být použity v aplikacích, které vyžadují delší klíč pro větší bezpečnost.

3.6.2 Haval

Haval je hashovací algoritmus, který byl navržen v roce 1992 Y. Zhengem, J. Pieprzykem aj. Seberrym. Poslední úpravy na tomto algoritmu byly provedeny v roce 1997, kdy byla nalezena kolize algoritmu s délkou klíče 128 bitů. Od té doby nebyl nahlášen ani jeden úspěšný útok na tyto funkce. Algoritmus podporuje 15 různých úrovní zabezpečení, které se liší délkou klíče (128/160/192/224/256 bitů) a volitelným počtem výpočetních cyklů funkce. Jako výchozí je považován algoritmus s délkou klíče 256 bitů a 5 cykly výpočtu.

3.6.3 Tiger

Tiger byl navržen v roce 1995 R. Andersonem a E. Bihamem. Tato funkce je speciálně vyvinuta pro 64 bitové platformy. Velikost výstupu je 192 bitů. Má dvě delší verze, vytvářející 128 a 160 bitový výstup, známé jako Tiger/128 a Tiger/160, kde je původní hash zkrácena na požadovanou délku.

3.6.4 Whirpool

Tato hashovací funkce byla navržena V. Rijmenem a P. S. L. M. Barretem. Funkce byla postupně vyvíjena a měla tři verze. První, Whirpool-0 byla podrobena analýze v projektu NESSIE. Druhou vylepšenou verzí byla Whirpool-T, která byla zahrnuta do portfolia kryptografických primitiv tohoto projektu. V této druhé verzi byla ovšem nalezena chyba, která se týkala schopnosti rozptylu výsledků funkce. Ta byla posléze opravena a výsledná hashovací funkce nese jednoduché jméno Whirpool. Tato výsledná funkce byla též přijata jako standard organizací ISO pod označením ISO/IEC 10118-3:2004.

3.6.5 Grindahl

Poprvé představena na konferenci FSE (Fast Software Encryption) v roce 2007. Autory jsou L.R. Knudsen, Ch. Rechberger a S. S. Thomsen. Funkce je založená na blokové šifře Rijndael. Funkce Grindahl existuje ve dvou variantách, a to jako Grindahl-256 a Grindahl-512. Funkce jsou zpracovány stejným způsobem a jsou odlišné jen velikostí hashovacího klíče.

Základním principem je rozdělení na bloky o velikosti 4x13 bytů, S tímto polem bloků jsou dále prováděny následující transformace:

- SubBytes – nelineární substituční funkce převzatá ze specifikace šifry Riijndael
- ShiftRows – cyklicky posouvá byty řádku v závislosti na jeho délce
- MixColumns – také převzata z šifry Rijndael a provádí transformaci sloupců
- AddRoundKey – také původem z Rijndael, může být zaměněna s funkcí AddConstant, která vnáší asymetrii do každého cyklu změnou posledního bytu pole

Nakonec je připojeno zakončení, podobně jako u MD5, a je provedeno dalších 8 cyklů funkce, které zajišťují dostatečné rozprostření hashovacího klíče. [12]

4 Hash Array Mapped Trie

The Hash Array Mapped Trie (HAMT), hashovaci pole mapované do stromové struktury, je založené na principu hash tabulky. Vypočítaný hash z klíče nám tedy složí pro adresování v rámci této struktury a nalezení hodnoty přiřazené tomuto klíči. Array Mapped Trie AMT, pole mapované do stromu, slouží k efektivnímu implementování této požadované struktury. AMT je univerzální datová struktura, která je často využívána jako základ k současným algoritmům v mnoha aplikacích. Algoritmy stromů byly původně vytvořeny panem Fredkinem v roce 1960 a poté vylepšeny pány Bentleyem a Sedgewickem (1997) jako The Ternary Search Trees (TST) a následně pány Nillson a Tikkanen (1998) jako Level Path Compressed Trees (LPC). AMT jsou ale výkonnostně 3 - 4x rychlejší než TST, spotřebovávají o 60 % méně místa a zároveň jsou také rychlejší než LPC stromy.

Během vyhledávání jsou bity progresivně využívány z hashů k procházení stromu, dokud není nalezen klíč nebo hodnota. Během vkládání jsou úrovně AMT postupně rozšiřovány používáním vícera bitů z hash hodnoty, dokud není nová hashovaná hodnota odlišena od hodnoty, která se již ve struktuře nachází.

Základní princip HAMT, rozdělování na základě hashů, byl vyvinut a zkombinován s koncepčními základy Lineárního Hashování (LH), aby algoritmus dosahoval efektivnější výkonnosti. Lineární hashování od Litwina, Neimata a Schneidera z roku 1993 je vyvinuto z Dynamic Hash Tables (Larson 1988), nabízí efektivnější řešení kolizí a nižší zátěž na úložiště při zachování akceptovatelného faktoru načítání a přerozdělování při kolizních stavech. Nový Partition Hash algorithm (PH) funguje na principu rozdělování bloků rovnoměrně mezi daný počet záznamů a používá sdružené bloky ke sdílení za účelem vylepšení načítacího faktoru. Rozdělená hashovaná hodnota je udržována v každém bloku a je klíčem pro vkládání a vyhledávání ke konkrétnímu bloku. To vede k jednotnému přístupu při vyhledávání a náročnost vkládání je velice nízká pro vysoké hodnoty načítacího faktoru. Algoritmy pak mohou být různě optimalizovány podle potřeby. Záleží tak na požadavku, co pro daný algoritmus je důležitější, jestli se upřednostní rychlost vkládání před hledáním, či naopak.

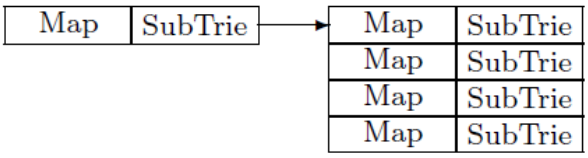
4.1 Základy Array Mapped Trie, AMT

Především by mělo být zmíněno, že všechny algoritmy, které následují, byly optimalizovány pro 32bit architekturu. Tudíž má AMT implementace základní kardinalitu o velikosti 32.

Strom je reprezentován nodem a počtem arků vedoucích k podstromu a každý ark reprezentuje mnoho různých následujících podstruktur. Aby se zachovala struktura pro 32bit architekturu, musíme omezit kardinalitu arků na 32, tedy od 0 do 31. Hlavní dilema při reprezentaci takových stromů je uhádnout adekvátní rovnováhu mezi požadovanou rychlostí průchodu a minimalizování ztrát místa v paměti u prázdných arků.

Celočíselná bitmapa je použita k reprezentování existence všech 32 možných arků a asociační tabulka obsahuje ukazatele k požadovanému podstromu nebo koncovým nodům. Jeden bit v bitmapě reprezentuje platný ark, zatímco nula ark prázdný. Ukazatele v tabulce jsou uchovávány v setřízeném stavu a zodpovídají za správné pořadí každého bitu v bitmapě. Obrázek 6 znázorňuje základní princip

struktury. Je třeba poznamenat, že mapovací hodnoty navazují do dalších neprázdných tabulek, ve kterých se nachází další vrstvy podstromu.



Obrázek 6 - Základní princip HAMT [1]

Hledání arku pro symbol „s“, požaduje nalezení svého požadovaného bitu v bitmapě a poté spočítání následujících bitů hashi klíče k získání indexu do setřízeného podstromu. Tento proces hledání arku má na starost funkce CTPOP (implementace viz Kód 2 - Implementace CTPOP). Dnešní CTPOP (Count Population) instrukce jsou dostupné na většině moderních počítačových architekturách, včetně Intel Itanium, Compaq, Alpha, Motorola Power PC, Sun UltraSparc a další, případně mohou být emulovány s pomocí nevolatilních referenčních posunů ke spočtení bitů v bitmapě.

```
GetElementBitIndex( bitIndex, bitMapHash)
{
bitsToCheck = bitMapHash << (maxChildrenForNode - bitIndex - 1) << 1;

        // Start of Popcount operation CTPOP
bitsToCheck = bitsToCheck - ((bitsToCheck >> 1) & 0x55555555);
bitsToCheck = (bitsToCheck & 0x33333333) + ((bitsToCheck >> 2) & 0x33333333);
return (((bitsToCheck + (bitsToCheck >> 4)) & 0x0F0F0F0F) * 0x01010101) >> 24;
}
```

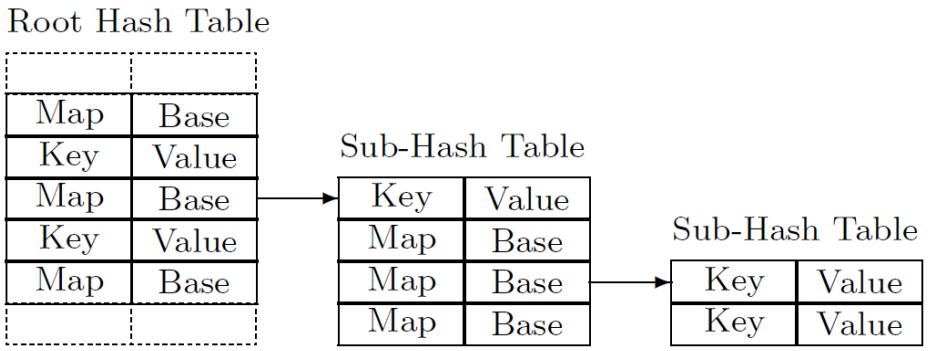
Kód 2 - Implementace CTPOP

4.2 Implementace HAMT

Základní pointa spočívá v počáteční velikosti kořenové hashovací tabulky a následném vyrovnávání stromu, podle toho, jak roste set ukládaných klíčů k udržení optimální výkonnosti. Čím větší velikost hashované tabulky kořene stromu, tím méně času je nutné strávit řešením kolizí. HAMT předpokládá, že hashovaná tabulka má nekonečnou velikost a používá AMT k zastoupení řídkých polí. V čase vkládání klíče je hashovací funkce použita k vygenerování prefixu indexu pole. Nejdůležitější bity hashe jsou používány k reprezentaci nejdůležitějších bitů[8] indexovaného pole. Použitím AMT struktury je hash spotřebováván při každém následujícím vnoření k podstromu, dokud není nalezen listový node s klíčem. Pokud je odlišný, tak se nový klíč přidá tak, že se odstraní současný klíč, vypočte se jeho rozšířený hash

[8] Z překladu The Most Significant Bit

a přidají se oba kolidující klíče do nového podstromu. Tento proces přidávání podstromů se opakuje tak dlouho, dokud mezi sebou dva klíče již nekolidují. Další bity z hashe jsou postupně dopočítávány.



Obrázek 7 - Realizace průchodu HAMT [1]

Realizace tohoto průchodu datovou strukturou HAMT je zobrazena na obrázku výše (viz Obrázek 7). Je složen z kořenové hashovací tabulky o velikosti 2*t*, kde *t* typicky začíná na 5 a každá položka je kořenem nodu hashované tabulky podstromu s velikostí 32. Každá položka v hash tabulce je buď klíč a hodnota nebo AMT ukazatel a bitmapa.

4.2.1 Hledání klíče

Je třeba spočítat celý 32 bit hash pro klíč, vzít z něj *t* nejdůležitějších bitů a použít je jako celočíselný index v kořenové hash tabulce. V tomto stavu mohou nastat 3 případy.

1. Položka je prázdná, ukazující, že klíč není v hash stromu.
2. Položka je pár klíče/hodnoty a klíč buď souhlasí s hledaným klíčem, což indikuje úspěch a pokud ne, tak chybu.
3. Položka má 32 bitmapovou subhash tabulku a ukazatel na podstrom[9]. Ta ukazuje na setřízený list položek v neprázdné subhash tabulce.

Dále je třeba vzít následujících 5 bitů z hashe a použít je jako celočíselný index do bitmapy. Pokud je nulový, tak položka v hash tabulce je prázdná a indikuje chybu, jinak je to jedna, a tedy je třeba spočítat následující bity použitím CTPOP a výsledek použít jako index do neprázdné položky v listu u báze. Tento proces je opakován (přibíráním dalších 5 bitů z hashe) dokud není nalezen koncový pár klíče/hodnoty nebo hledání selže. Typicky je k nalezení potřeba pouze malé množství iterací a je důležité zmínit, že klíč je porovnáván pouze jednou, a to při nalezení klíče koncového nodu. Tímto tento proces přispívá k rychlosti prohledávání struktury, protože je vynecháno mnoho zbytečných dotazů do paměti. Odlišnosti jsou jednoduše a včasně detekovány a mnohdy ani není třeba porovnávat klíče mezi sebou.

[9] Někdy známý jako báze.

AMT datová struktura je používána k minimalizování prázdných hodnot v tabulce. Jakmile strom roste, tak je zde malá, ale stále narůstající pravděpodobnost, že počet operací způsobí, že 32bit hash bude vypotřebován a bude muset být vytvořen nový.

Za předpokladu, že hash funkce generuje náhodnou distribuci klíčů, pak v průměru bude přes hash klíč nalezen koncový node po *log N* bitech. S AMT je každou iterací vzato 5 bitů, z čehož vyplývá náročnost hledání $\frac{1}{5} log\, N$ resp. *O(log N)*. Tato náročnost může být snížena v ideálním případě až na *O(1)*.

```
KeyValueNode Search(key)
{
rootShift = (bitShift * (startingLevelCount + 1 - root.BitmapHash))
currentBitHash = key.GetHashCode()
rootBitmapIndex = currentBitHash & (root.Nodes.Length - 1) //získaní indexu hash hodnoty pro adresaci
linkerNode = root.Nodes[rootBitmapIndex] //vrátí existující LinkerNode na daném indexu
if (linkerNode != null)//pokud existuje linkerNode tak hash koliduje a musíme jít do podstromu
{
   currentBitHash >>= rootShift    //bitový posun na další index
   for (cLevel = 1 cLevel != root.BitmapHash + 1 keyBitHash >>= bitShift, cLevel++)
   {  //hierarchické prochazení HAMT s bit posuny
      indexGivenBitIndex = CountPop(currentBitHash & mask, linkerNode.BitmapHash)
         //CTPOP pro získání daného indexu z hashe
      keyValueNode = linkerNode.Nodes[indexGivenBitIndex] //pokus o nalezení klíče
      if (keyValueNode != null)   //klíč nalezen
      {
         return comparer.Equals(keyValueNode.Key, key) ? keyValueNode : null
      }      //porovnání nalezeného klíče s testovaným
      else    //byl nalezen podstrom
      {
         linkerNode = linkerNode.Nodes[indexGivenBitIndex]   //vnoření do podstromu
      }
   }
   foreach (keyValueNode in linkerNode.Nodes)  //procházení koncového listu
   {
      if (comparer.Equals(keyValueNode.Key, key))
      {  //pokud se klíč shoduje, vrátí jeho hodnotu
         return keyValueNode
      }
   }
} else  //klíč se nachází přímo v kořenu
{
   return root.Nodes[rootBitmapIndex]
}}
```

Kód 3 - Algoritmus hledání klíče

4.2.2 Vkládání

Prvotní kroky pro přidání nového klíče do hash stromu jsou velice podobné jako u vyhledávání. Postupuje se jako u vyhledávacího algoritmu, dokud se nenarazí na jednu ze dvou následujících příčin.

1. Je nalezena prázdná pozice v hash tabulce.
2. Existuje odkaz na sub-hash tabulku.

V tomto případě, pokud se to stalo v kořenové hash tabulce, tak nový pár klíče/hodnoty je umístěn na prázdné místo v kořenové tabulce. V případě sub-hash tabulky je nutné přidat nový bit do bitmapy a zvětšit velikost sub-hash tabulky o jedna. Nová sub-hash tabulka musí být alokována, existující podtabulka do ní zkopírována, nový pár klíče/hodnoty je nutné přidat do sub-hash tabulky v setřízeném pořadí a starou hash tabulku vyprázdnit. Jinak by starý klíč kolidoval s nově přidaným. V každém případě současný klíč musí být vyměněn sub-hash tabulkou a získat následujících 5 bitů z hashe. Pokud stále nastává kolize, tak je tento proces opakován, dokud žádná kolize nenastává. Existující klíč je pak vložen do sub-hash tabulky a nový klíč je uložen. Pokaždé jak je použito dalších 5 bitů z hashe, tak se snižuje pravděpodobnost kolize o faktor 1:32. Mohou ale nastat případy, u kterých je spotřebován celý 32 bitový hash a musí být vytvořen hash nový k odlišení 2 různých klíčů.

```
void Add(key, value)
{
   KeyBitHash = key.GetHashCode()
KeyBitIndex = KeyBitHash & (rootCount - 1) //získaní indexu hash hodnoty pro adresaci
KeyValuePair = KeyValueNode(key, value)
linkerNode = root.Nodes[KeyBitIndex]   //vrátí existující LinkerNode na daném indexu
if (linkerNode != null)    //pokud existuje linkerNode tak hash koliduje a musíme pokračovat do podstromů
   {
      KeyBitHash >>= rootBitShiftAmount      //bitový posun na další index
      for (cLevel = 1 cLevel != root.BitmapHash + 1 KeyBitHash >>= bitShift, cLevel++)
      {        //hierarchické prochazení HAMT s bit posuny
         KeyBitIndex = KeyBitHash & maskAmount
         indexGivenBitIndex = CountPopAlgorithm(KeyBitIndex, linkerNode.BitmapHash)
         //CTPOP pro získání daného indexu z hashe
      if (IsBitSetPosition(KeyBitIndex, linkerNode.BitmapHash)) //test obsazenosti indexu
         {
            nextNodeKey = linkerNode.Nodes[indexGivenBitIndex]
               //pokus o získání kolidujícího páru klíč/hodnota
         if (nextNodeKey != null)    //klíče kolidují
            {
               KeyBitHash >>= bitShift   //bit posun
            ReplacePairToLinkNode(linkerNode.Nodes, nextNodeKey, KeyValuePair, cLevel)
               //dva kolidující Nody jsou vloženy do podstromu o úrovni cLevel
         } else
            {
```

```
                linkerNode = linkerNode.Nodes[indexGivenBitIndex]
            }               //získání dalšího linkerNodu
            } else
            {
              InsertIntoLinkerNode(linkerNode, KeyValuePair, KeyBitIndex)
        }               //uložení hodnoty na prazný index

        }
        AppendLeavesToBucket(linkerNode, KeyValuePair)    //vložení linkerNodu do bloku
    }
      keyValueNode = root.Nodes[KeyBitIndex] //pokus o získání kolidujícího páru klíč/hodnota z kořene
stromu
    if (keyValueNode != null)// pokud není null je nalezen pár, který by kolidoval se současným
      {
        KeyBitHash >>= rootBitShiftAmount
      ReplacePairToLinkNode(root.Nodes, nextNodeKey, KeyValuePair, cLevel = 0)
      //nastává kolize dvou párů u kořene. Obě hodnoty se přesunou do podstromu
    } else
      {
       root.Nodes[KeyBitIndex] = KeyValuePair   //uložení hodnoty na daný index kořene
    }}
```

Kód 4 - Algoritmus vkládání klíče

4.2.3 Alokace paměti

Z citace Nilssona a Tikkanena: "Performance critically depends on the memory allocation system and careful organization is rewarded" [13], v překladu tedy, výkonnost kriticky závisí na systému alokování paměti a opatrné hospodaření s ní bude odměněno. Jelikož sub-hash tabulka má rozsah od 1 do 32 položek, tak požadavek pro nové hash tabulky je postupné rozložení hodnot napříč celým rozsahem, v našem případě tedy rozsahem o velikostí 32. Tohoto rozložení je dosaženo tak, že se uchovává pole 32 propojených seznamů prázdných sub-hash tabulek. První položka v poli obsahuje seznam všech volných sub-hash tabulek o délce jedna, druhá seznam všech sub-hash tabulek o délce dva a stejným principem to dále pokračuje. Navíc alokování nové sub-hash tabulky vyžaduje minimum času, buď odebráním jedné tabulky z volného seznamu nebo alokováním nové tabulky z volného paměťového zásobníku (memory pool). Paměťový zásobník je alokován ze systémové paměti v blocích. Uvolněná sub-hash tabulka je přidána do požadovaného volného seznamu, připravena k naplnění.

Správného rozložení se ale velice špatně dosahuje. Je třeba dbát na správnou vyváženost struktury, neboť s tím, jak roste set klíčů, tak je uvolňováno více a více sub-hash tabulek, které následně mohou chybět. Pokud se nebude kontrolovat uvolňování sub-hash tabulek, tak to povede k velkému nárůstu spotřeby paměti. Typicky se spotřeba může pohybovat až k 3násobku potřebné velikosti. Díky blokům paměťového zásobníku ale mohou být defragmentovány ke zpětnému obnovení ztraceného místa. Hranice defragmentace je nastavena jako procento ze spotřebované paměti, která byla celkově uvolněna.

Pokud je tato hranice překročena, tak blok paměťového poolu bude defragmentován během následujícího procesu vkládání klíče. S touto změnou je spotřeba volného místa omezena na pár procent s minimálním dopadem na čas vkládání klíčů.

Když je vkládán nový klíč, tak je podtabulka uvolněna o *x* hodnot. Ty jsou uloženy na dané místo do volného listu a do nové podtabulky je potřeba *x+1* hodnot, které z ní byly odebrány. Jelikož po *n* vkládáních je *n* podtabulek uvolněno, a tudíž je vyžadováno *n* nových podtabulek. Aby bylo dosaženo konstantní úrovně volné paměti, tak defragmentace musí obnovit v nejhorším případě *n(1-f)* nodů stromu, pokud nemají potřebnou velikost (kde *f* značí zlomek alokované paměti a spotřebované paměti po *n* přidaných klíčích). Každý blok tedy v průměru obsahuje zlomek *1-f* podtabulek a *f* podtabulek prázdných.

4.2.4 Algoritmus vyrovnávání (Resize)

V určitém bodě jakmile hash strom roste, je velice vhodné provést vyrovnání celé root hash tabulky, tedy resizing. Dá se ukázat, že tato operace není příliš časově náročná, v nejhorších případech se pohybuje časové náročnosti vkládání či vyhledávání.

Každý sub-hash strom reprezentuje 32 hodnot, které mohou být zaplněné či prázdné. Pokud je každý kořen stromu přerozdělen, tak *2(t+5)* hodnot sub-hash tabulek má své odpovídající místo v nové kořenové hash tabulce. Hodnoty z neprázdné sub-hash tabulky musí být zkopírovány do nové root hash tabulky a staré sub-hash tabulky jsou uvolněny. Není však třeba provádět nové hashování hodnot. Pravděpodobnost, že bude třeba provést nové spočtení hashe klíče k uložení do přerozdělené struktury je téměř nulová. Ale i tak by se na tento fakt měl brát zřetel a nezanedbat jej.

Celá hash tabulka kořenu může být zpracována najednou nebo po částech. To znamená, že jednotlivé sub-hash tabulky budou přemístěny do nové kořenové tabulky až budou potřeba. Nejdříve je vytvořena nová tabulka kořenu se všemi položkami nastavenými jako prázdné. Upravený vyhledávací algoritmus porovnává prvních *t* bitů hashe s indexem vyvážení. Pokud je nižší, tak vyhledávání používá starou hash tabulku jinak vyhledávání začne v nové hash tabulce kořenu a vezme *t + 5* bitů z hashe k utvoření indexu a dále vyhledávání pokračuje dle normálního postupu. Algoritmus vkládání je upraven obdobně.

Hash tabulka by měla být vyrovnána, když nově vytvořená hash tabulka zaujme pouze určitý zlomek celkového alokovaného místa v paměti, řekněme $\frac{1}{f}$. To znamená, že prvních $\frac{1}{5} log \frac{N}{f}$ přístupů bude vyměněno jediným přístupem do hash tabulky. Tudíž průměrná časová náročnost vyhledávání a vkládání bude $\frac{1}{5} log N - \frac{1}{5} log \frac{N}{f}$ nebo $\frac{1}{5} log f$, což vychází na *O(1)*. Vyrovnávání tabulky proběhne pouze tehdy, pokud nová tabulka přesahuje 32x tabulku starou nebo když původní tabulka se zmenší na velikost $\frac{1}{32} f$. Je tedy vhodné přesunout hodnoty ze staré tabulky do nové tabulky kořenu jednou za zhruba $32 \frac{f}{2}$ operací vložení k dokončení tohoto procesu předtím, než bude potřeba provést další cyklus vyrovnání celé struktury. Jelikož přesunutí hodnoty z tabulky kořenu je ekvivalentní k několika průměrným časům při procesu vkládání, tak tyto případy mají za následek nejpomalejší časy při procesu vkládání. Pokud se dopředu může odhadnout velikost celého HAMT, tak se mu může přizpůsobit velikost hash

tabulky kořenu. Větší velikostí počáteční hash tabulky zhoršujeme sice výkon struktury na menším počtu záznamů, ale čím více se HAMT zaplňuje, tím se trend výkonnosti otáčí. S větší počáteční hash tabulkou také rostou nároky na spotřebu paměti. Vyšší rychlost je dána tím, že se sníží počet procesů vyrovnávání stromu.

```
void RootResize()
{
   newRoot = LinkerNode(root.Nodes.Length * maxChildrenForNode, root.BitmapHash - 1)
      //vytvoření nového kořene HAMT se základní tabulkou N-krát větší
   rootShiftAmount = (bitShiftAmount * (startingLevelCount + 1 - root.BitmapHash))
      //určení délky bitů pro bit posuny
   for ( i = 0; i != root.Nodes.Length; i++)
   {  //procházení všech hodnot v kořenové tabulce
      currentRootChild = root.Nodes[i]
      currElementIndex = 0

      for ( j = 0; j != 32; j++) //maximální počet položek pro podstromy
      {
         if (((currentRootChild.BitmapHash >> j) & 1) == 1)
         {     //zjištění pozice v novém podstromu
            currentNode = currentRootChild.Nodes[currElementIndex++]
            newRoot.Nodes[(j << rootShiftAmount) | i] = currentNode
               //vložení Nodu do nové struktury
         }
      }
   }
   root = newRoot
}
```

Kód 5 - Algoritmus vyrovnávání stromu

4.2.5 Využívání paměti

S algoritmem vyrovnávání hash tabulky kořenu je spotřeba paměti konstantním násobkem *N*. Když budeme předpokládat, že při procesu vyrovnání tabulky bude faktor vyrovnání nastaven na 4, tak pro každou lokaci v průměru přísluší 3 klíče, které se hashují na dané místo. Každý z těchto případů od 0 až do *N* se řídí Poissonovým rozdělením. Podle toho se dá zjistit, že v průměru 1,8 % míst bude prázdných, 7,3 % lokací bude mít pár klíče/hodnoty a zbytek bude patřit pro hodnoty v podstromech. Poté když se vyberou případy 2 do *N*, tak průměrný počet podstromů, které obsahují další podstromy, je možné také spočítat obdobným způsobem. Zjistíme tedy, že v průměru 18,8 % míst bude mít následující podstrom, který bude následován dalším podstromem. Z těchto hodnot můžeme zhruba odhadnout, kolik místa

nám struktura bude zabírat paměti. Požadované místo v paměti tedy bude vyjádřeno jako počet párů klíče/hodnoty jako *N* a počet odkazů na podstromy ve struktuře bez počtu záznamů v počáteční hash tabulce kořenu. Tedy *N+0,25N*(1+0,188-0,073)*, což bude vycházet přibližně kolem *1,28N*.

4.2.6 Odebrání klíče

Proces odebrání klíče ze struktury může obnášet pár komplikací, nejdůležitější jsou ale dvě následující. Pokud sub-hash strom obsahuje více než 2 hodnoty, tak je klíč odstraněn a pozice, na které se nacházel, se označí prázdnou proměnnou. To vyžaduje, aby nová menší tabulka mohla být alokována a stará smazána k uvolnění místa. Druhý případ nastane, pokud jsou v tabulce pouze dvě hodnoty, pak je zbývající hodnota přemístěna do sub-hash tabulky rodiče a současná tabulka uvolněna.

Jo možné nastavit i hranici pro free memory pool, pokud je překročena, pak alokované sub-hash tabulky v pool bloku jsou přesunuty do volného místa k ostatním blokům. Odebrané hodnoty jsou odstraněny z volných listů a pool blok uvolněn ze systémové paměti.

4.2.7 Hash funkce

Dobře propracovaná hash funkce může vysoce ovlivnit celkovou výkonnost HAMT struktury. Pro celý HAMT, stejně tak hash tabulky je použit Universal hash. Tato hash funkce byla vyvinuta k vytváření 32 bitových hashu. Algoritmus vyžaduje, aby mohl být hash rozšiřován o libovolný počet bitů. Tohoto je docíleno tak, že se pomocí rehashovaní zkombinuje klíč s integerem, který udává stupeň stromu, hodnota 0 je dedikována pro kořen stromu. Jestliže dva klíče vracejí stejný hash hned v první iteraci, pak se musí spočítat rehash na vyšší stupeň stromu. V takovém případě klesá následná pravděpodobnost kolizí mezi těmito dvěma klíči o 2^{32}, pro třetí úroveň 2^{64}, čtvrtou 2^{96} … atd. S účinnou hash funkcí by měly být hashe unikátní po průměrně log *n* bitech, takže k použití rehash funkce nedochází příliš často.

5 Experimentální část

V této kapitole provedu sérii různých testů, díky kterým získáme lepší přehled o tom, jak se mezi sebou výkonnostně liší vybrané datové struktury. Veškeré výkonnostní testy byly provedeny v prostředí jazyka C# v režimu sestavení x64. Pokud by bylo použito sestavení v režimu x32, velice brzy bychom narazili na strop omezení RAM[10] pro 32 bit procesy[11]. Hardwarové specifikace testovacího stroje jsou uvedeny v tabulce níže (viz Tabulka 3).

PROCESOR	INTEL® CORE™ I5-3470 3,2GHZ
RAM	2x8GB DDR3 1600MHz
ÚLOŽIŠTĚ	SSD Crucial MX100 256 GB
OPERAČNÍ SYSTÉM	Microsoft © Windows 10 Pro

Tabulka 3 - Hardwarové specifikace testovacího stroje

Pro srovnání jsem použil struktury dostupné pro C# prostředí z knihovny System.Collections.Generic a strukturu na bázi stromu (Efficient Text Pattern Search Tree [2]). Z této C# kolekce jsem vybral jednu z nejzákladnějších struktur pro ukládání řetězců, což je třída List. Jelikož se věnujeme tématu hashovacích funkcí, tak do srovnání musím také přidat strukturu HashTable. A jako posledním kandidátem byla zvolena třída Dictionary, která patří mezi nejefektivnější struktury pro práci s řetězci. U hash tabulky jsem se rozhodl tuto třídu dále rozdělit na dva podtypy. V prvním případě jsem ukládal do hash tabulky na pozici klíče přímo hodnotu řetězce jako string. A v případě druhém jsem použil vlastní řešení spočítání hashe a ten jsem následně použil jako klíč do hash tabulky.

Všechny algoritmy jsem otestoval na třech rozdílných řetězcích hashovacích funkcí. MD5 o délce 8 znaků, SHA-1 o délce 10 znaků a SHA-512 o délce 32 znaků. Pro lepší přehlednost budou zobrazeny grafy těchto tří hashovacích funkcí jen na struktuře HAMT a Dictionary. Pro srovnání s ostatními algoritmy jsem použil hash funkci SHA-512, a to z důvodu větší délky řetězce, neboť se snáze projeví odlišnosti mezi jednotlivými strukturami.

Velikost sady testovacích řetězců jsem zvolil od 100 tisíc prvků až do 40 milionů. Tato horní hranice byla zvolena z důvodu hardwarového omezení RAM paměti. Algoritmy jsou ale různě náročné na RAM paměť, a proto jsem byl schopen u struktury stromu otestovat maximálně 2 miliony záznamů. V případě hash tabulky jsem se dostal na limit struktury po 20 milionech záznamech.

5.1 Vkládání

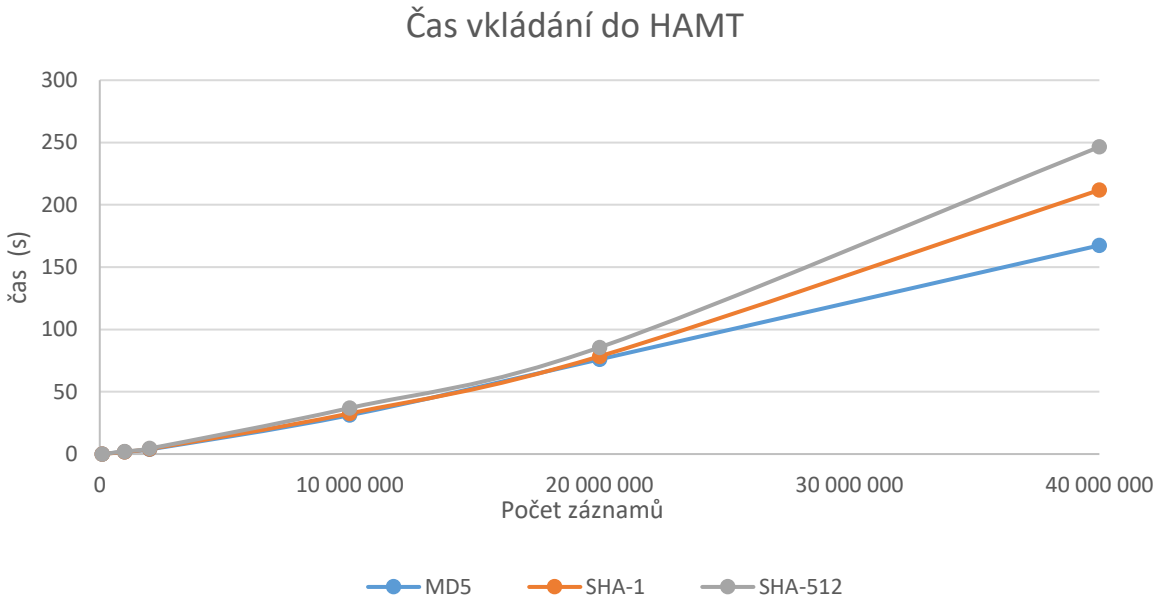
Proces vkládání nebývá většinou tím nejdůležitějším požadavkem pro hledaní efektivní struktury pro práci s řetězci. Je to z toho důvodu, že většina služeb funguje na postupném zvětšování struktury a má-

[10] Random Access Memory
[11] Plyne z omezení architektury, 2^{32} B ≈ 4 GB

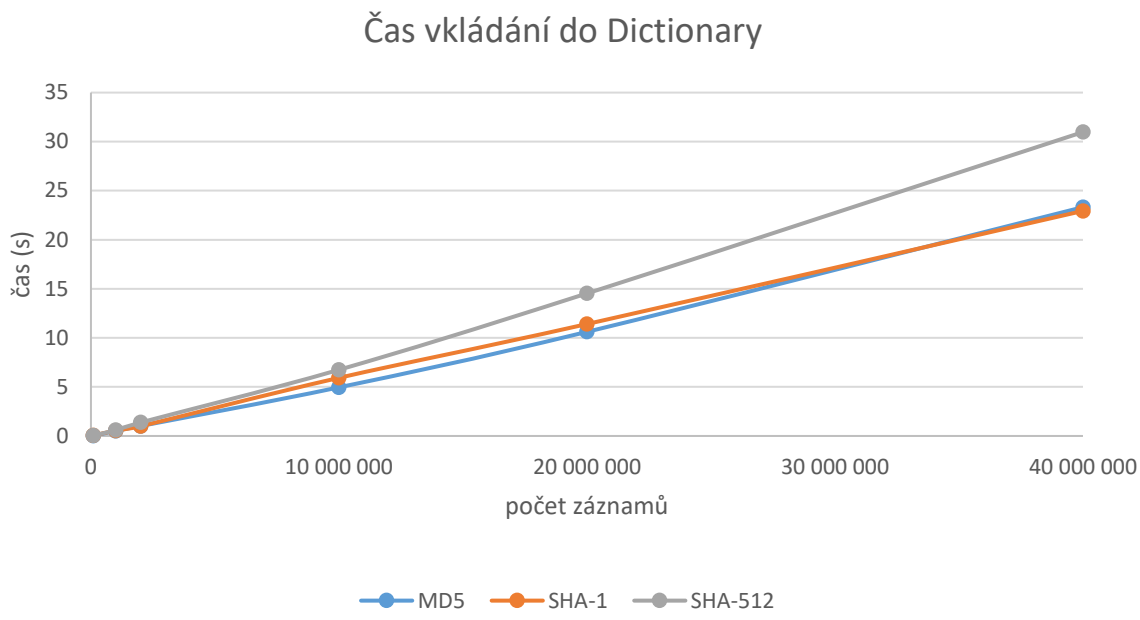
lokdy se stane, že je třeba využít plného potenciálu rychlosti vkládání. Přesto by sestavení takové struktury nemělo trvat neúměrně dlouho dobu. Většinou platí, že pro lepší výkonnost pro proces hledání jsme ochotni obětovat část výkonu z procesu vkládání.

Z výsledku měření (viz Graf 2) můžeme zjistit, že pro strukturu HAMT je proces vkládání pro kratší řetězce popisován téměř lineární křivkou. S větším vstupním řetězcem rostou výrazně nároky na vkládání hodnot do struktury. Delší řetězec zabírá více místa, a jelikož probíhají procesy vyrovnávání stromu, tak přenášení delších řetězců v rámci struktury trvá delší dobu.

Vyšší náročnost na čas vložení nastává i v případě Dictionary. Ačkoli mezi MD5 a SHA-1 jsou rozdíly minimální, neboť se tyto hash funkce od sebe liší velice málo, tak použití hash funkce SHA-512 se projevilo už výrazněji. Zatímco u HAMT v případě použití SHA-512 se časy vkládání začaly odlišovat až při vyšších počtech záznamů, tak u Dictionary je tento trend více postupný. Dalo by se říct, že dle velikosti vstupního řetězce ukládaného do struktury je náročnost lineární s rostoucím počtem záznamů.

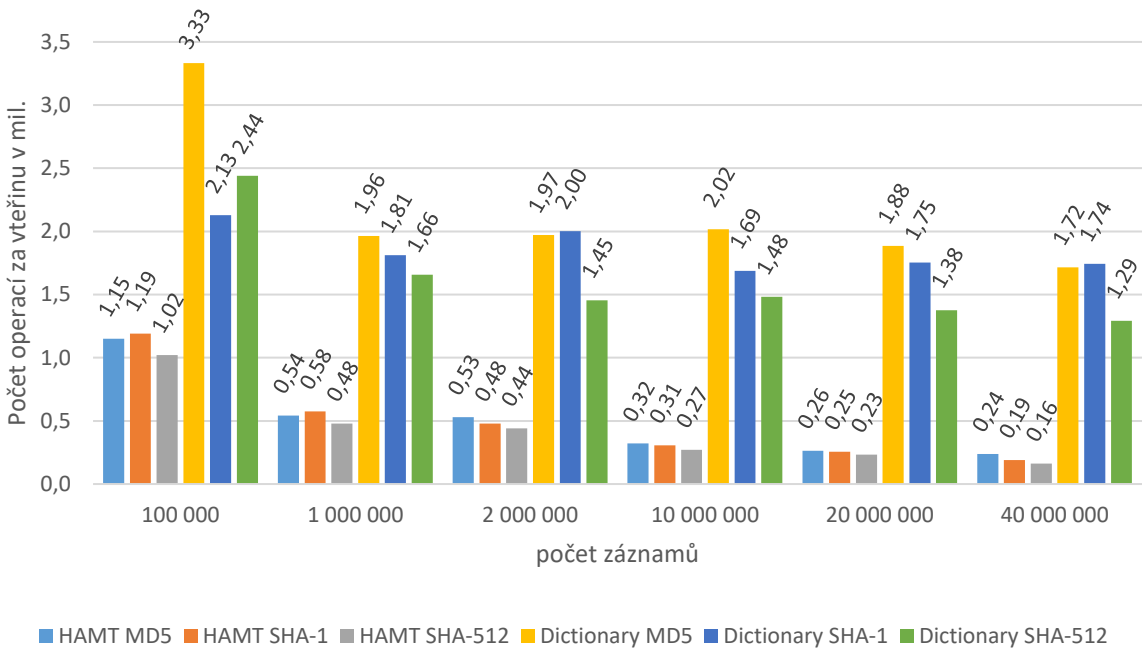


Graf 2 - Čas vkládání záznamů do HAMT pro MD5, SHA-1, SHA-512



Graf 3 - Čas vkládání záznamů do Dictionary pro MD5, SHA-1, SHA-512

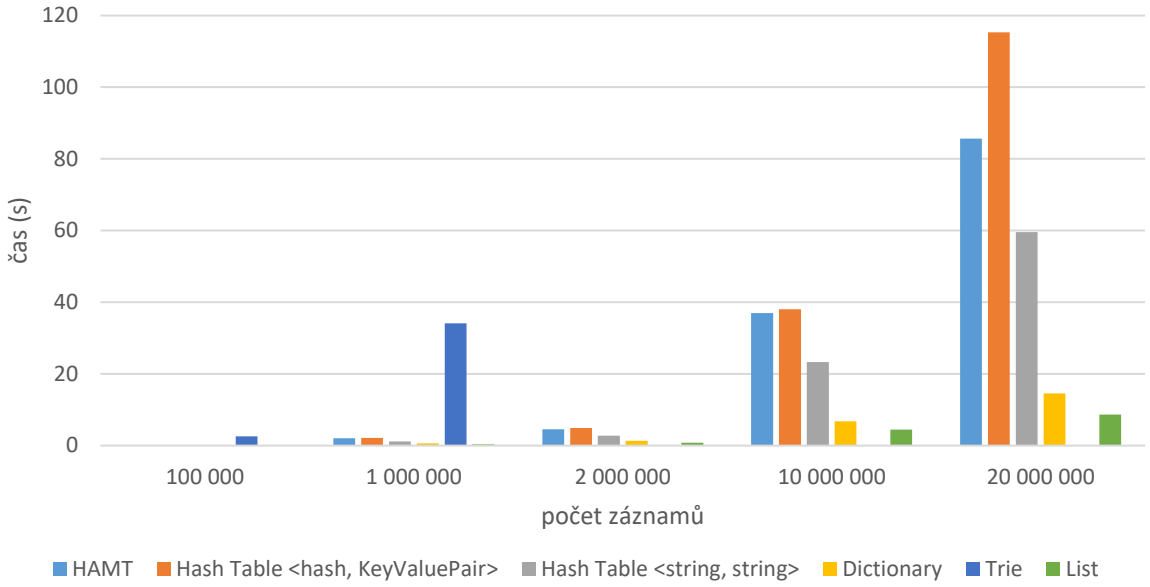
Lepší přehled o výkonnosti algoritmu v závislosti na použité hash funkci zjistíme na hodnotách propustnosti k operaci vkládání (viz Graf 4). U Dictionary je propustnost vkládání téměř konstantní. Sníženého výkonu začíná dosahovat až při vysokém koeficientu zaplnění struktury nebo v případě použití delšího řetězce. Propustnost vkládání u HAMT se také snižuje s větší délkou vstupního řetězce i v případě většího zaplnění struktury, což je způsobeno náročnějším procesem vyrovnávání struktury. Při tomto procesu nastává kopírování struktur v rámci paměti do nové vyvážené struktury stromu, takže kopírování větších struktur zabere více času a to se negativně ovlivní na čase vložení záznamu. Při větší zaplněnosti struktury se tento problém více prohlubuje, neboť se musí kopírovat nejen delší řetězce, ale i vícero podstruktur v rámci HAMT.



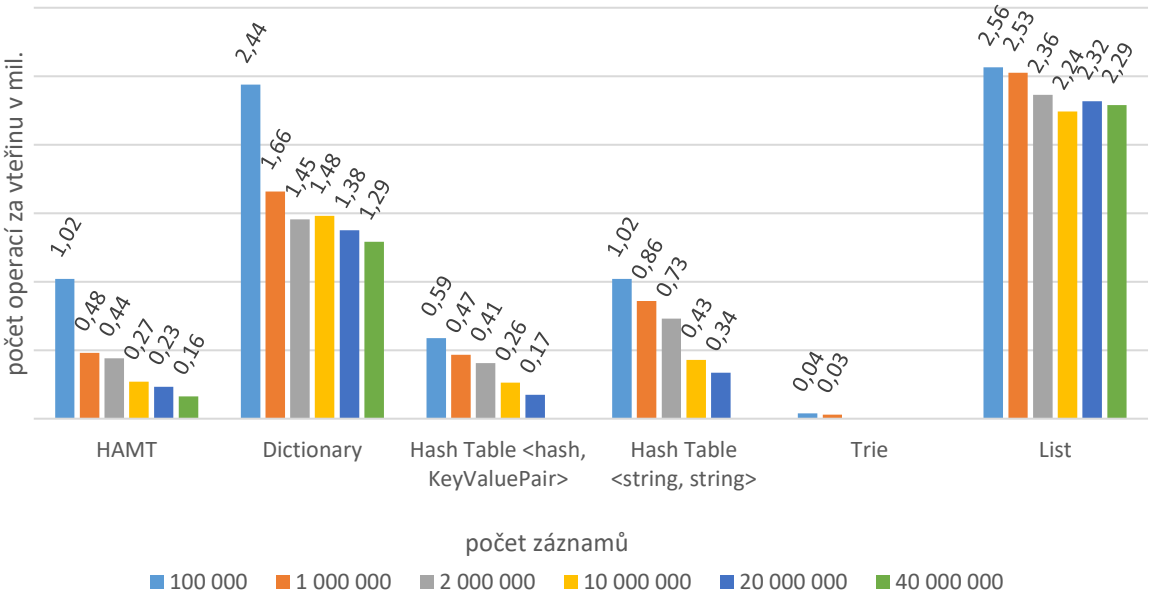
Graf 4 - Propustnost vkládání HAMT a Dictionary pro MD5, SHA-1, SHA-512

Když porovnáme jednotlivé struktury mezi sebou podle časů vkládání (viz Graf 5), tak zjistíme, že nejrychlejší strukturou pro vkládání hodnot je List. Není to však překvapením, neboť List nepoužívá žádné složité indexační algoritmy a ukládá vstupní řetězec na konec pole. Díky tomuto principu je propustnost operací i pro velké sady hodnot téměř konstantní. Naproti tomu se struktura stromu ukázala jako značně neefektivní pro ukládání většího množství řetězců a její výkonnost tak s rostoucí sadou strmě klesá. V porovnání s ostatními strukturami je propustnost vkládání na velice špatné úrovni o hodnotě několika desítek tisíc operací za vteřinu (viz Graf 6).

Druhou nejefektivnější strukturou se ukázala třída Dictionary, která si byla schopna zachovat vysokou propustnost operací i při velkém počtu ukládaných záznamů. HAMT a Hash Table mají podobné časy vkládání a jejich propustnost s rostoucím počtem záznamů klesá obdobně. Zajímavostí je odlišnost mezi dvěma typy Hash tabulky. Implicitní hashovací funkce tabulky pracuje zřejmě efektivněji a to se projevilo zhruba dvounásobně vyšší propustností vkládání.



Graf 5 - Časy vkládání (HAMT, Hash table, Dictionary, Trie, List)

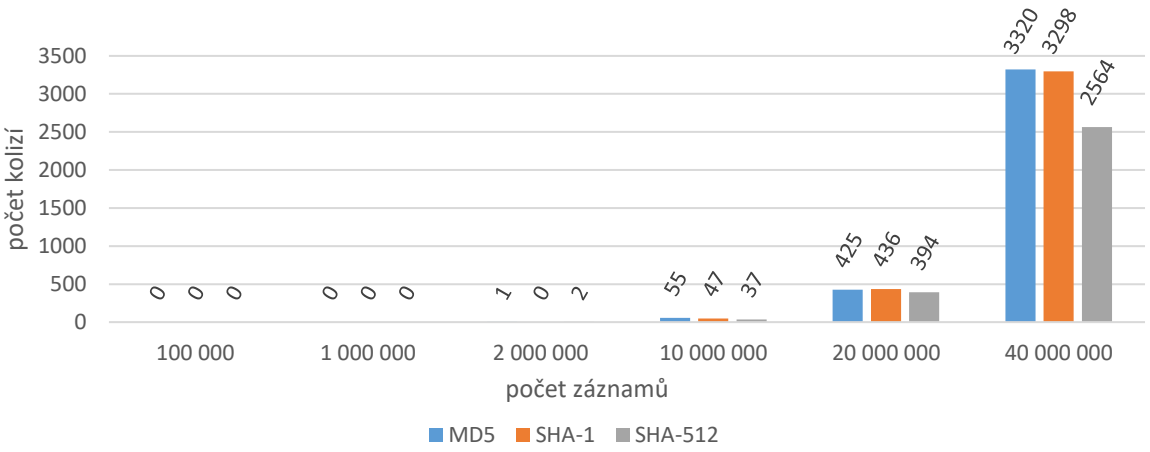


Graf 6 - Propustnost vkládání

5.2 Výskyt kolizí

Tento test je značně ovlivněn typem použité hashovací funkce. V případě HAMT struktury se začínáme s kolizemi potýkat kolem 2 milionech záznamů. Výskyt kolizí strmě stoupá s vyšším počtem ukládaných záznamů. V případě delšího řetězce SHA-512 je kolizních stavů o něco méně, je to ale na úkor vyšších nároků na paměť a procesu vkládání. Nejvíce kolizních stavů jsem naměřil u Hash tabulky s explicitní metodou počítání hashe (viz Tabulka 4). Jelikož jsem nepoužil alternativní řešení uložení klíče do hash

tabulky, než s pomocí prvotně spočteného hashe, tak tyto kolizní záznamy nebyly do hash tabulky uloženy. V případě Hash tabulky s implicitní hashovací funkcí a Dictionary nedochází k žádným kolizním stavům, se kterými by si tyto kolekce neporadily. Mají tedy úspěšnost hledání klíče 100 %, stejně jako List a Strom. Ty ale nepoužívají hashovací funkce pro indexaci řetězců, takže se jich přímo tento problém netýká.



Graf 7 - Kolizní stavy v HAMT pro MD5, SHA-1, SHA-512

V následující tabulce (viz Tabulka 4) je zobrazen přehled zjištěných kolizních stavů pro vlastní implementaci spočítání 64 bitového hash klíče a funkcí GetHashCode, která je dostupná prostředí C#. Má implementace int64 hash funguje na principu rozdělení vstupního řetězce na dvě části. Pro tyto části se spustí běžná funkce GetHashCode spojí je do jednoho dlouhého řetězce a ten se následně uloží do datové struktury int64. Díky této snadné operaci se nám sníží počet kolizních stavů o polovinu. Nevýhodou jsou ale větší nároky na paměť, neboť datový typ int64 neboli long zabírá na jeden záznam 8 bytů, kdežto int32 pouze byty 4.

	POČET KOLIZNÍCH HASHŮ	
POČET ZÁZNAMŮ	C# hash funkce	vlastní int64 hash
100 000	10	2
1 000 000	120	57
2 000 000	443	225
10 000 000	11464	5844
20 000 000	46639	23192

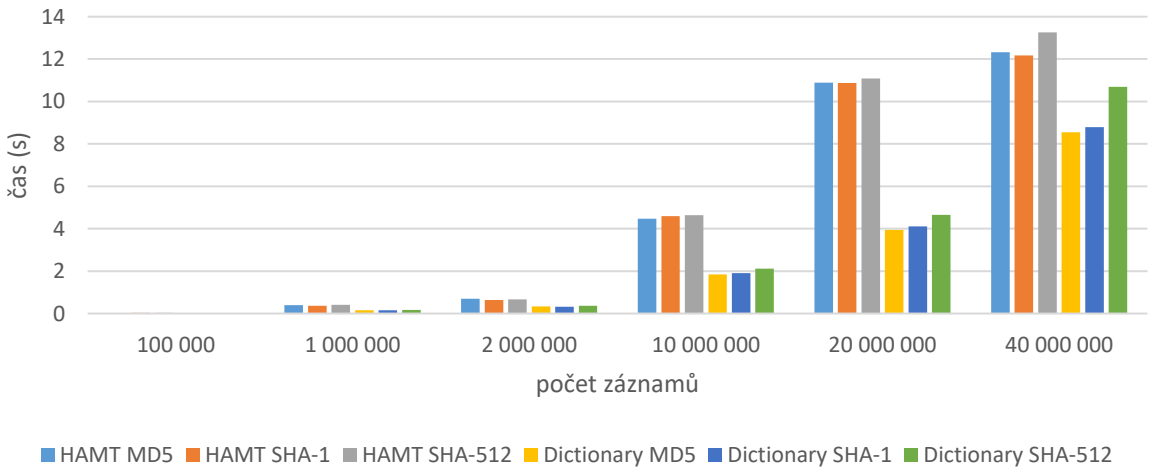
Tabulka 4 – Srovnání počtu kolizních hashů dle použité funkce

5.3 Vyhledávání

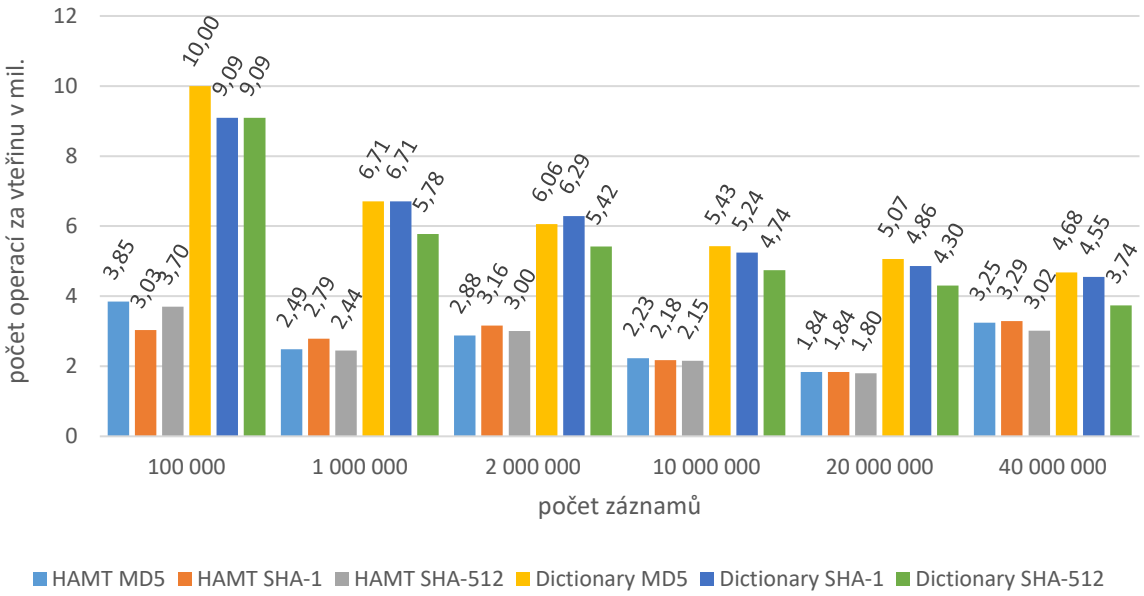
Vyhledávání patří mezi nejdůležitější ukazatele při výběru struktury pro rychlé indexování textu. Měření probíhalo na principu naplnění struktury daným počtem záznamů a následném dotazování se do struktury na každý ukládaný klíč. Tím se zároveň ověřilo, zda struktura je schopna uložit veškeré množství vstupních záznamů.

Z naměřených výsledků zjistíme (viz Graf 8), že délka řetězce u HAMT nehraje větší roli při procesu vyhledávání. Časy vyhledávání jsou pro všechny tři hash funkce více méně podobné a liší se od sebe pouze minimálně. Rozdíl může být způsoben nejpravděpodobněji horším rozložením dat ve struktuře a jejím mírně složitějším procházením. U Dictionary je rozdíl minimální v případě hash funkce MD5 a SHA-1, ale větší délka řetězce SHA-512 se u této struktury již začíná projevovat hlavně u vyššího počtu záznamů.

Propustnost vyhledávání se u HAMT pohybuje kolem 2-3 miliónů operací za vteřinu. S narůstajícím počtem záznamů postupně klesá, ale pouze do doby než je spuštěn proces vyrovnání stromu, který zvětší velikost kořenové tabulky. Velikost kořenové tabulky by se dala vhodně určit při inicializaci, pokud bychom předem znali počet záznamů, se kterými bude HAMT pracovat. Pokud by byla zvolena příliš velká tabulka kořenu, projevilo by se to na zhoršeném času vyhledávání záznamů. Naopak v případě postupného zvětšování tabulky kořenu při vyrovnávání stromu se zhorší čas u procesu vkládání, ale bude zachována co nejlepší možná rychlost vyhledávání při postupném zaplňování HAMT. V případě propustnosti u struktury Dictionary se vyšší počet záznamů projevuje na výkonnosti mnohem více. Její propustnost a tedy i výkonnost s rostoucí strukturou výrazně klesá. Dalo by se tedy předpokládat, že pokud bych nenarazil na hardwarové omezení testování struktur, tak HAMT v určitém bodě bude dosahovat lepší výkonnosti při vyhledávání, než Dictionary.



Graf 8 - Časy vyhledávání HAMT a Dictionary pro MD5, SHA-1, SHA-512

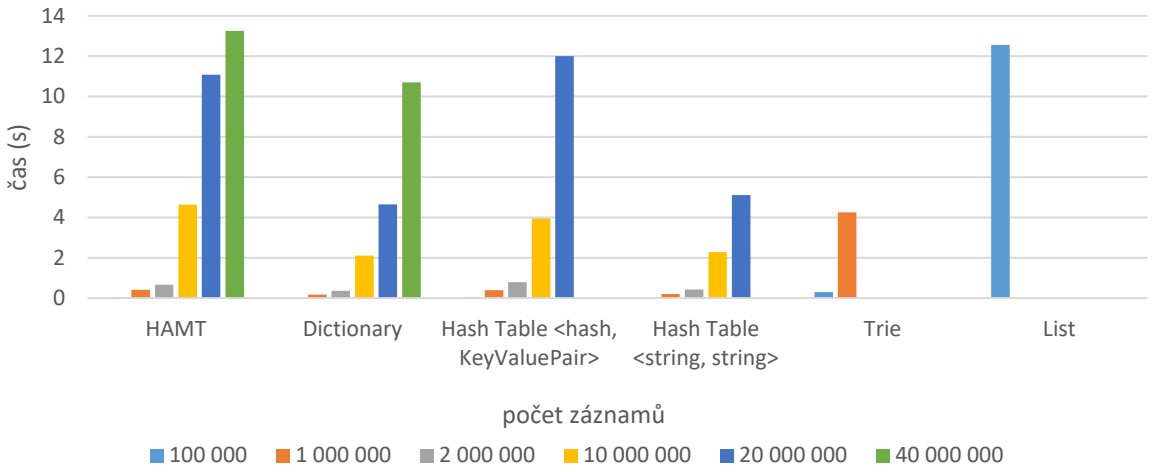


Graf 9 - Propustnost vyhledávání HAMT a Dictionary pro MD5, SHA-1, SHA-512

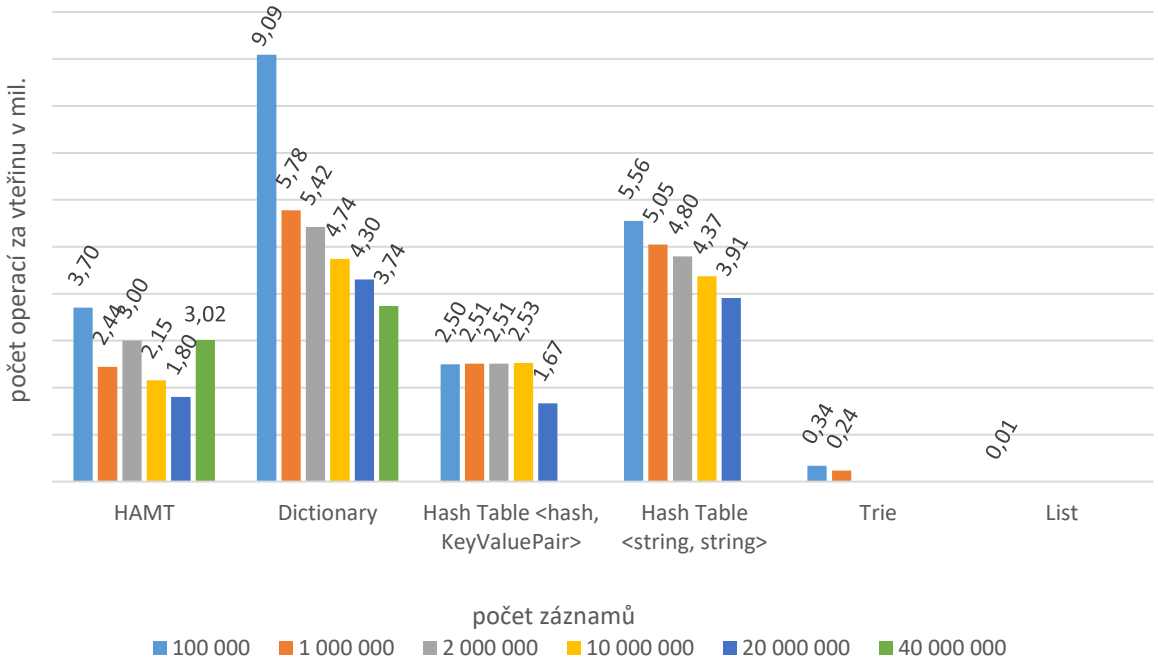
Při srovnání HAMT s ostatními strukturami, je na první pohled zřejmé, že List je naprosto nevhodný pro dohledávání dat. Kvůli své náročnosti se dal otestovat pouze na nejmenší sadě testovaných záznamů. Struktura stromu si vedla o poznání lépe než List, ale s ostatními indexovanými strukturami se rovnat také nemůže.

Z grafu časů vyhledávání (viz Graf 10) se jeví jako nejrychlejší struktura pro vyhledávání dat Dictionary, následovaná Hash tabulkou s implicitní hash funkcí. Hash tabulka s explicitní hash funkcí pracuje méně efektivně a pohybuje se na úrovni rychlosti HAMT.

Lepší představu o rychlosti vyhledávání ale zjistíme z Graf 11, kde je zobrazena propustnost vyhledávání pro jednotlivé struktury, dle velikosti struktury. U Dictionary i Hash tabulky s implicitní funkcí hash, propustnost postupně klesá s rostoucí velikostí struktury. V případě Hash tabulky s explicitní funkcí hash je trend propustnosti konstantní a výrazně padá, až při vysokém koeficientu zaplnění struktury.



Graf 10 - Časy vyhledávání pro HAMT, Dictionary, hash table, Trie, List



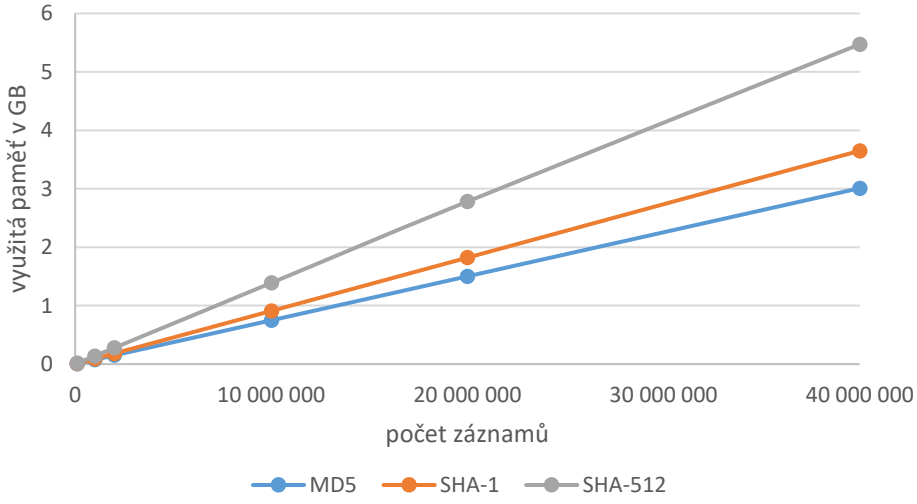
Graf 11 - Propustnost vyhledávání pro HAMT, Dictionary, hash table, Trie, List

5.4 Paměťová náročnost

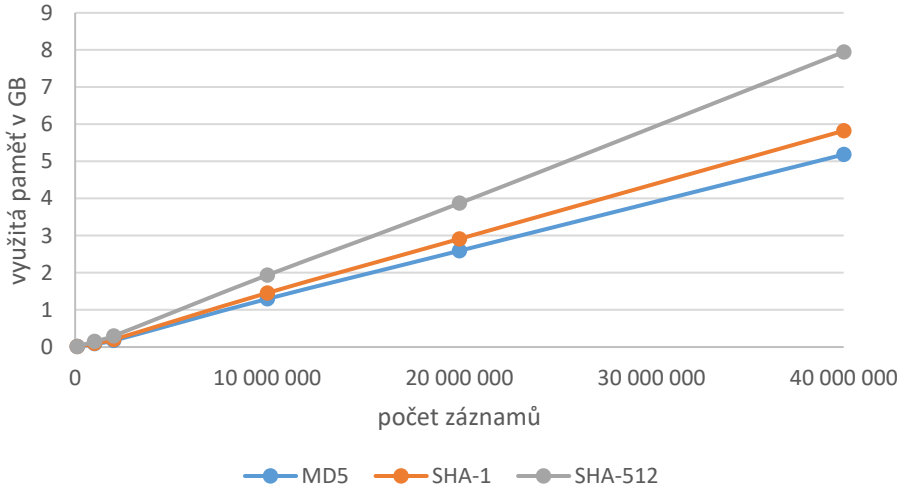
Náročnost algoritmů na spotřebu paměti vychází z toho, jak efektivně je algoritmus schopen využít hash klíč jako index na pozici do dané struktury. Pokud budeme vycházet z paměťové náročnosti Listu, do kterého se ukládají hodnoty v nezměněném stavu, tak jakýkoli nižší výsledek znamená, že ukládané hodnoty se do struktury komprimují.

Důvod, proč jsem strukturu stromu nebyl schopen otestovat pro vyšší počet záznamů, byla právě její enormní náročnost na paměť. Už při 1 milionu záznamů přesahovala spotřeba paměti 8 GB. Je to z důvodu velké neefektivity ukládání řetězců. Pro každý node stromu náleží pouze jeden symbol, a tudíž paměťová náročnost tohoto algoritmu roste exponenciálně.

U HAMT dosahovala komprese poloviny původní velikosti řetězce. Na Graf 12 je zobrazeno srovnání paměťové náročnosti vstupních hash funkcí pro strukturu HAMT. Ačkoli délka řetězce pro funkci MD je pouze 8, tedy třetinová proti SHA-512 s délkou řetězce 32, tak její nároky na paměť jsou zhruba poloviční. U Dictionary jsou nároky na paměť vyšší než u HAMT (viz Graf 13), která je k využívání paměti šetrnější. Při menším počtu záznamů dosahuje poměrně dobrého kompresního poměru oproti Listu, avšak s vyšším počtem záznamů rostou i nároky pro indexování záznamů pro jejich efektivnější hledání. U vyššího počtu záznamů Dictionary dosahuje minimální úspory místa. Náročnost na paměť dle použité hash funkci je podobná jako v případě HAMT.

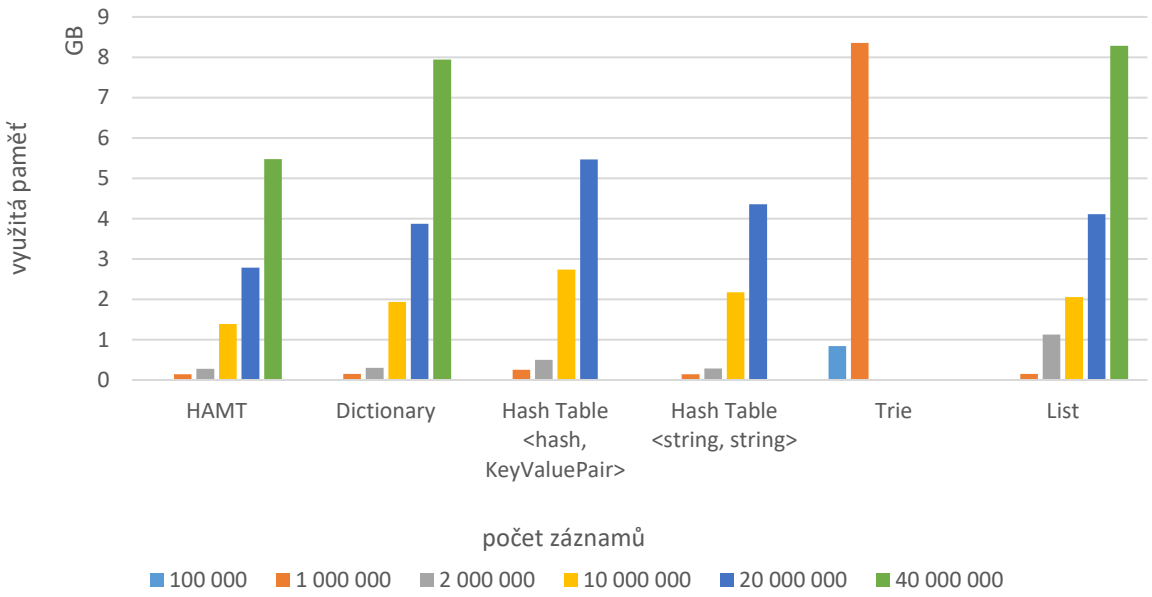


Graf 12 - Paměťová náročnost HAMT pro MD5, SHA-1, SHA-512



Graf 13 - Paměťová náročnost Dictionary pro MD5, SHA-1, SHA-512

Paměťová náročnost ostatních algoritmů více méně kopíruje paměťovou náročnost Listu (viz Graf 14), není to ale dané tím, že by tyto struktury nepoužívaly kompresní metody. Jejich náročnost je závislá na tvorbě indexů pro efektivnější průchod strukturou za účelem rychlejšího dohledávání dat. Důkazem je paměťová náročnost u nižšího počtu záznamů v těchto strukturách. Na menší sadě záznamů není nutné tvořit tak složité procesy indexování, které s rostoucí strukturou výrazně nabývají na své rozsáhlosti. To negativně ovlivňuje spotřebu paměti a v případě delších vstupních řetězců to s sebou přináší omezení v celkovém počtu záznamů, které do této struktury můžeme uložit. V mém případě u použití SHA-512 to bylo u Hash tabulky kolem 25 miliónů záznamů a Dictionary dosahovalo svého limitu při zhruba 45 milionech záznamů.



Graf 14 - Paměťová náročnost datových struktur HAMT, Dictionary, hash table, Trie, List

6 Závěr

V této práci byl nejprve popsán teoretický koncept stringologie se zaměřením na vyhledávání řetězců. Tyto řetězce vycházejí z hashovacích funkcí, které díky svým vlastnostem umožňují efektivnější a jednodušší práci s řetězci. Ukázal jsem výhody jejich využití a zdůraznil problémy, které tyto funkce můžou naopak přinést a jak těmto problémům předcházet.

V praktické části jsem se zaměřil na implementaci datové struktury Hash Array Mapped Trie. Popsal teoretický princip, jak tuto datovou strukturu vytvořit podle autora Phila Bagwella a pro nejdůležitější části kódu (vkládání a hledání řetězce, vyrovnávání struktury) ukázal pseudokódy na jejich implementaci.

V experimentální části této práce jsem tuto implementovanou strukturu HAMT porovnal s dalšími strukturami používanými pro indexování dat. Z prostředí C# se jednalo o třídy ze System.Collections.Generic, jmenovitě Dictionary a Hash table a strukturou stromu implementovanou A.P. Castañem. Mezi tyto struktury sice úplně nepatří třída List, ale bylo zajímavé ostatní struktury s ní porovnat z pohledu časů vkládání a paměťové náročnosti.

Z testů jsem zjistil, že struktura stromu je velice neefektivní pro ukládání a vyhledávání většího počtu řetězců, a tak vypadla z možného doporučení, zda tuto strukturu využít pro rychlé indexování textu. V případě ostatních datových struktur už výsledky nebyly tak jednoznačné. HAMT struktura společně s Hash tabulkou s explicitní hash funkcí jednoznačně zaostává v čase vkládání proti Hash tabulce s implicitní hash funkcí. Pokud nepočítám strukturu Listu, který pro vkládání jednoznačně překonává všechny ostatní struktury, tak nejlepších časů vkládání dosahuje struktura Dictionary. Zároveň ji šlo společně s HAMT otestovat na sadě počtu záznamů přesahujících 20 miliónů. V případě obou Hash tabulek jsem narazil na omezení počtu záznamů ve struktuře, a proto jejich testování skončilo na počtu 20 milionech záznamů. Třída Dictionary má také své omezení v počtu uložených záznamů, okolo 47 miliónů. HAMT struktura není přímo omezená počtem záznamů, které je do ní možné uložit, ale s vyšším počtem záznamů se potýká s vysokým nárůstem kolizních stavů. Z důvodu hardwarového omezení nedostatku paměti jsem strukturu HAMT otestoval na maximálně 40 milionech záznamech, stejně jako v případě Dictionary.

U testů vyhledávání řetězců ve struktuře byly výsledky podobné jako u testů zaměřených na vkládání. Třídy Dictionary i Hash tabulka s implicitní hash funkcí dosahovaly skvělých hodnot u menšího počtu záznamů ve struktuře, avšak s narůstající velikostí struktury jejich propustnost k procesu vyhledávání postupně klesá. Nedostaly se ale do stavu, kdy by byla propustnost nižší než v případě HAMT. Trend propustnosti sice naznačoval, že tento stav může nastat, ale kvůli uvedeným hardwarovým omezením jsem se do tohoto stavu nedostal. Výsledky Hash tabulky s explicitní hash funkcí a HAMT byly velice podobné. V případě Hash tabulky se ale dá spíše doporučit použití její implicitní hash funkce, která pracuje mnohem efektivněji a zároveň řeší kolizní stavy automaticky.

Implementovaná struktura HAMT na daném počtu testovaných záznamů nedosahovala lepších výsledků než například třída Dictionary. Pokud by ale testy byly provedeny na stroji s větší dostupnou pamětí, mohli bychom dojít k výsledku, kdy struktura HAMT předčí ve výkonnostních textech i třídu Dictionary a stane se tak ideální strukturou pro rychlé indexování textových dat.

7 Literatura

1. **Bagwell, Phil.** *Ideal Hash Trees.* 2001.

2. **Castaño, Arnaldo Pérez.** A More Efficient Text Pattern Search Using a Trie Class in .NET. *VisualStudio Magazine.* [Online] 2012. https://visualstudiomagazine.com/articles/2015/10/20/text-pattern-search-trie-class-net.aspx.

3. **M. Crochemore, W. Rytter.** *Jewels of Stringology.* Singapore : World Scientific Publishing Co. Pte. Ltd., 2002. 981-02-4782-6.

4. *Combinatorial Algorithms on Words.* **Galil, Z.** Berlin : -, 1985. Open problems in stringology, NATO ASI Series F.

5. **doc. Ing. Jan Holub, Ph.D.** *Stringologie, komprese dat a biologie.* Brno : VUTIUM, 2014. 1213-418X.

6. **M. A. Maniscalco, S. J. Puglisi.** An efficient, versatile approach to suffix sorting. *J. Exp. Algorithmics.* 2008.

7. **J., Pinkava.** *Hashovací funkce v roce 2004.* Praha : PVT a.s., 2004.

8. **What is a Public and Private Key Pair?** *SSL2BUY.* [Online] [Citace: 1. 4 2018.] https://www.ssl2buy.com/wiki/what-is-a-public-and-private-key-pair.

9. Chauvaud., N. Rogier and P. *The compression function of MD2 is not collision free.* Ottawa, Canada : Carleton University, 1995.

10. RFC family. *RFC-editor.* [Online] 1992. http://www.rfceditor.org/rfc.

11. NIST. ***FIPS*** *180-1 Secure Hash Standard.* 1993.

12. Knudsen L.R., Rechberger Ch., Tomsen S. S. *Grindahl - a family of hash functions.* Graz, Austria : autor neznámý, 2007.

13. Nilsson, Tikkanen. *"relaxed" LC-Trees.* 1998.

14. C. Allauzen, M. Crochemore, M. Raffinot. *Factor Oracle: A new structure for pattern matching.* Berlin : Springer-Verlag, 1999.

15. Dan, Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge : Cambridge university press, 1997.

7.1 Seznam použitých symbolů a zkratek

HAMT	Hash Array Mapped Trie
AMT	Array Mapped Trie
LH	Lineární hashování
PH	Partition hash algoritmus
CTPOP	Count population algoritmus
RAM	Random access memory
MD	Merkle-Damagard
SHA	Secure Hash Standard

7.2 Seznam obrázků

7.3 Seznam tabulek

7.4 Seznam grafů

7.5 Seznam kódů

8 Přílohy

počet hodnot	struktura	použitá hash funkce	čas naplnění struktury (s)	počet kolizí	čas vyhle-dávání (s)	spotřeba paměti
100 000	HAMT	MD5	0,087	0	0,026	7 523 696
1 000 000		MD5	1,843	0	0,402	75 230 520
2 000 000		MD5	3,772	1	0,694	150 625 676
10 000 000		MD5	31,209	55	4,476	749 362 096
20 000 000		MD5	76,011	425	10,891	1 501 485 196
40 000 000		MD5	167,47	3320	12,319	3 009 269 520
100 000		SHA-1	0,084	0	0,033	9 123 864
1 000 000		SHA-1	1,739	0	0,359	91 229 428
2 000 000		SHA-1	4,174	0	0,633	182 625 396
10 000 000		SHA-1	32,618	47	4,593	909 371 652
20 000 000		SHA-1	78,446	436	10,879	1 821 477 004
40 000 000		SHA-1	211,94	3298	12,169	3 649 270 808
100 000		SHA-512	0,098	0	0,027	13 924 264
1 000 000		SHA-512	2,085	0	0,409	139 231 260
2 000 000		SHA-512	4,531	2	0,666	278 626 420
10 000 000		SHA-512	36,968	37	4,641	1 389 363 632
20 000 000		SHA-512	85,626	394	11,087	2 781 490 476
40 000 000		SHA-512	246,515	2564	13,261	5 470 954 520
100 000	Dictionary	MD5	0,03	0	0,01	8 601 343
1 000 000		MD5	0,509	0	0,149	86 001 343
2 000 000		MD5	1,015	0	0,33	172 001 343
10 000 000		MD5	4,957	0	1,842	1 295 970 744
20 000 000		MD5	10,612	0	3,948	2 591 941 568
40 000 000		MD5	23,322	0	8,556	5 183 884 056
100 000		SHA-1	0,047	0	0,011	10 201 343
1 000 000		SHA-1	0,552	0	0,149	102 001 343
2 000 000		SHA-1	0,999	0	0,318	204 001 343
10 000 000		SHA-1	5,926	0	1,907	1 455 970 744
20 000 000		SHA-1	11,407	0	4,116	2 911 941 568
40 000 000		SHA-1	22,937	0	8,794	5 823 884 056
100 000		SHA-512	0,041	0	0,011	15 001 343
1 000 000		SHA-512	0,603	0	0,173	150 001 343
2 000 000		SHA-512	1,375	0	0,369	300 001 343
10 000 000		SHA-512	6,748	0	2,11	1 935 970 744
20 000 000		SHA-512	14,529	0	4,648	3 871 941 568
40 000 000		SHA-512	30,975	0	10,695	7 943 884 056
100 000	HashTable <string>	MD5	0,133	2	0,035	19 293 328
1 000 000		MD5	1,826	57	0,321	185 477 792

počet hodnot	struktura	použitá hash funkce	čas naplnění struktury (s)	počet kolizí	čas vyhle-dávání (s)	spotřeba paměti
2 000 000		MD5	4,207	225	0,717	380 255 080
10 000 000		MD5	35,938	5844	3,789	2 129 316 440
20 000 000		MD5	80,925	23192	7,858	4 188 375 456
40 000 000		MD5				
100 000		SHA-1	0,183	0	0,031	20 554 632
1 000 000		SHA-1	2,101	72	0,369	201 474 360
2 000 000		SHA-1	4,646	227	0,796	405 399 984
10 000 000		SHA-1	32,779	5805	3,892	2 254 974 696
20 000 000		SHA-1	85,628	23365	8,207	4 507 975 320
40 000 000		SHA-1				
100 000		SHA-512	0,17	1	0,04	25 831 656
1 000 000		SHA-512	2,139	66	0,398	249 472 200
2 000 000		SHA-512	4,935	248	0,796	501 384 552
10 000 000		SHA-512	38,062	5925	3,955	2 734 670 136
20 000 000		SHA-512	115,305	23290	12,004	5 466 870 000
40 000 000		SHA-512				
100 000	HashTable <int>	MD5	0,058	0	0,013	7 700 240
1 000 000		MD5	1,015	0	0,174	77 000 240
2 000 000		MD5	2,009	0	0,365	154 000 240
10 000 000		MD5	21,345	0	2,002	1 535 949 912
20 000 000		MD5	51,682	0	4,188	3 071 900 616
40 000 000		MD5				
100 000		SHA-1	0,065	0	0,014	9 300 240
1 000 000		SHA-1	1,017	0	0,174	145 486 456
2 000 000		SHA-1	2,347	0	0,385	186 000 240
10 000 000		SHA-1	18,476	0	1,937	1 695 949 912
20 000 000		SHA-1	49,586	0	4,306	3 391 900 616
40 000 000		SHA-1				
100 000		SHA-512	0,098	0	0,018	14 100 240
1 000 000		SHA-512	1,163	0	0,198	141 000 240
2 000 000		SHA-512	2,739	0	0,417	282 000 240
10 000 000		SHA-512	23,291	0	2,288	2 175 949 912
20 000 000		SHA-512	59,622	0	5,114	4 351 900 616
40 000 000		SHA-512				
100000	Trie	MD5	1,835	0	0,356	408 271 256
1 000 000		MD5	19,996	0	3,076	4 003 730 688
2 000 000		MD5	47,514	0	6,684	7 948 337 184
100 000		SHA-1	1,864	0	0,223	517 246 992
1 000 000		SHA-1	22,372	0	3,471	5 091 668 984
100 000		SHA-512	2,609	0	0,296	843 459 072

počet hodnot	struktura	použitá hash funkce	čas naplnění struktury (s)	počet kolizí	čas vyhle-dávání (s)	spotřeba paměti
1 000 000		SHA-512	34,091	0	4,25	8 355 702 792
100 000	List	MD5	0,039	0	11,558	13 848 704
1 000 000		MD5	0,386	0		136 388 736
2 000 000		MD5	0,583	0		272 777 344
10 000 000		MD5	4,125	0		1363886720
20 000 000		MD5	8,356	0		2727773440
40 000 000		MD5	16,956	0		5455446880
100 000		SHA-1	0,038	0	12,556	15 448 704
1 000 000		SHA-1	0,388	0		152 388 736
2 000 000		SHA-1	0,782	0		304777472
10 000 000		SHA-1	4,255	0		1523887360
20 000 000		SHA-1	8,359	0		3047774720
40 000 000		SHA-1	17,108	0		6095549440
100 000		SHA-512	0,039	0		19 582 533
1 000 000		SHA-512	0,396	0		205 008 452
2 000 000		SHA-512	0,846	0		414 352 804
10 000 000		SHA-512	4,458	0		2 051 156 846
20 000 000		SHA-512	8,633	0		4 108 435 582
40 000 000		SHA-512	17,463	0		8216871164

Tabulka 5 - Výsledky měření